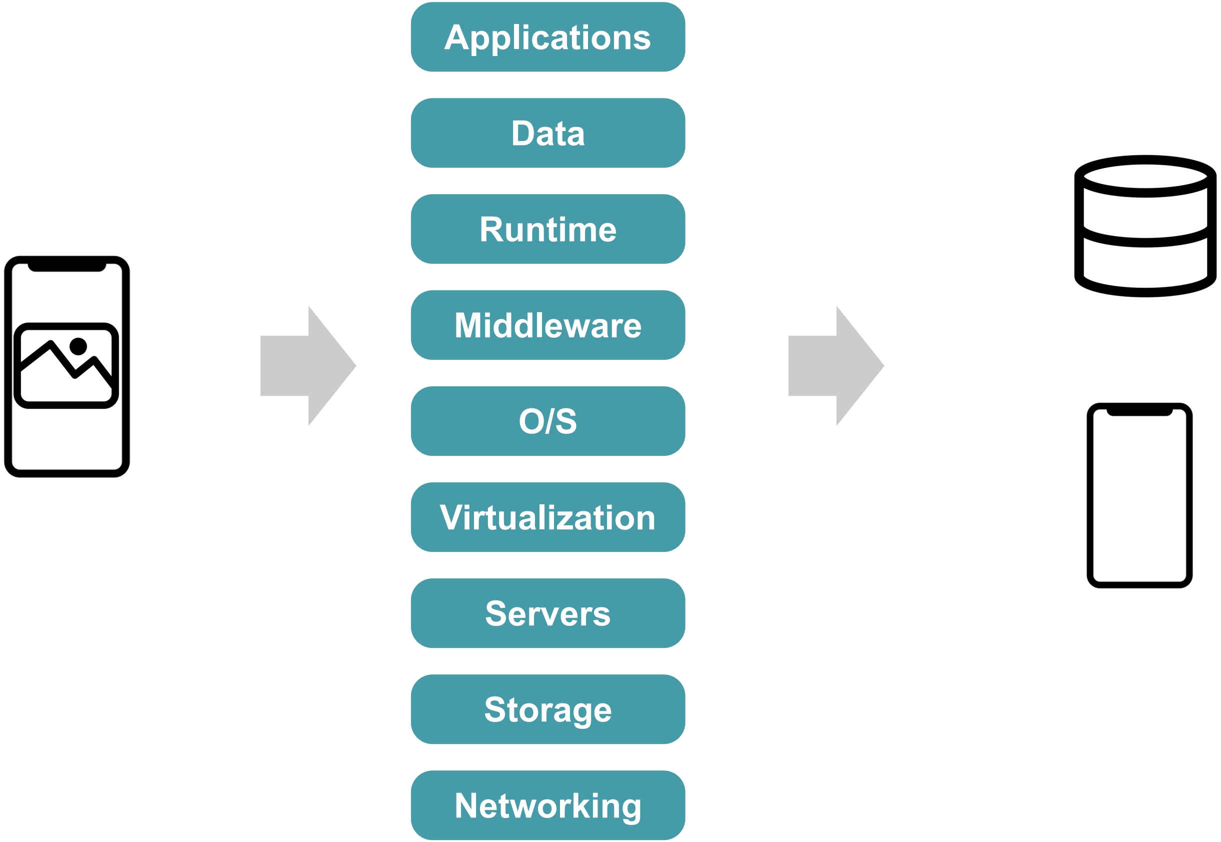
Netherite: Efficient Execution of Serverless Workflows

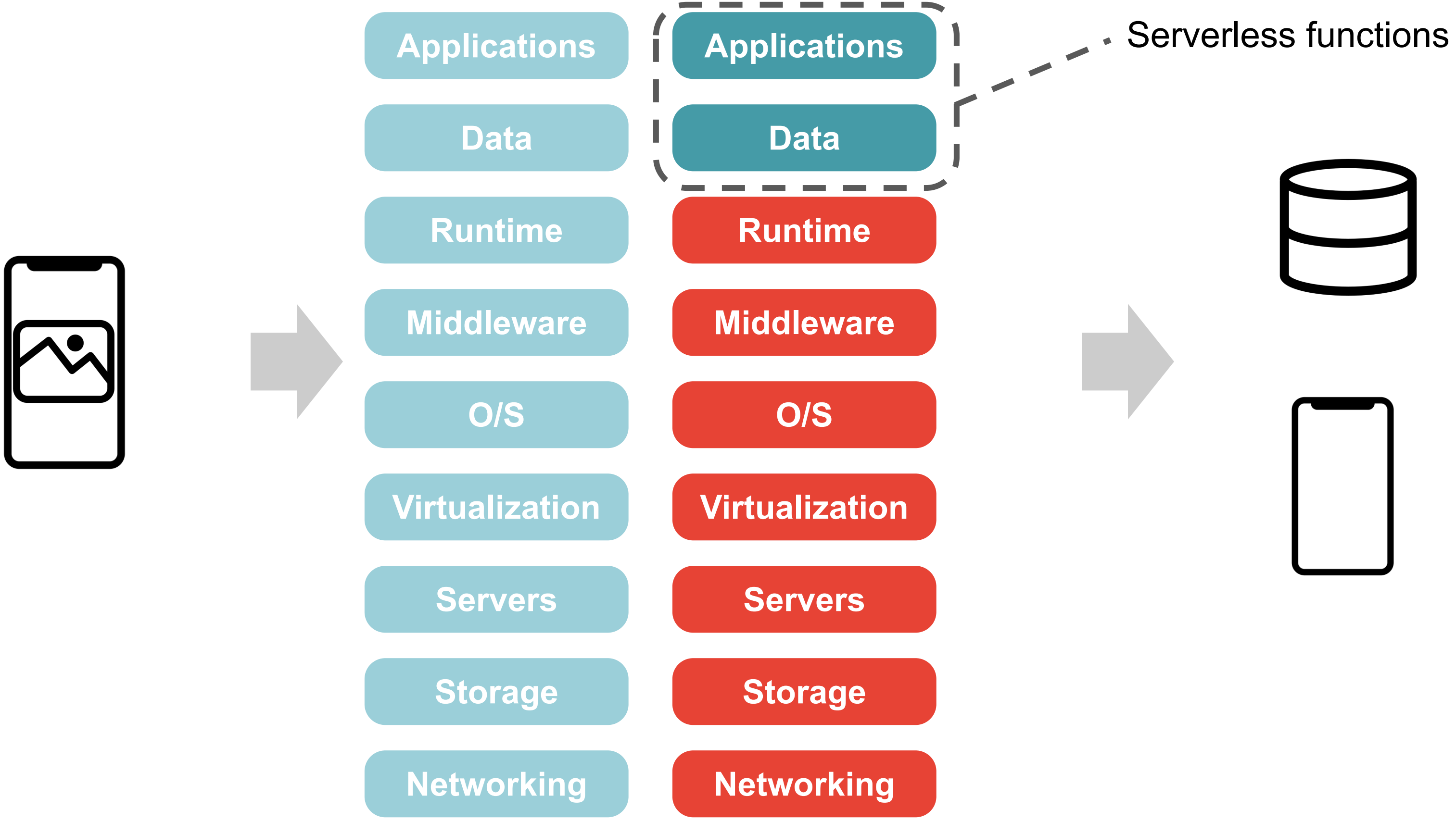
Wangyi Chen, Zhenhuan Wu

BOSTON UNIVERSITY

User Story With A Server



User Story With Serverless



Applications

Data

Runtime

Middleware

O/S

Virtualization

Servers

Storage

Networking

Applications

Data

Runtime

Middleware

O/S

Virtualization

Servers

Storage

Networking

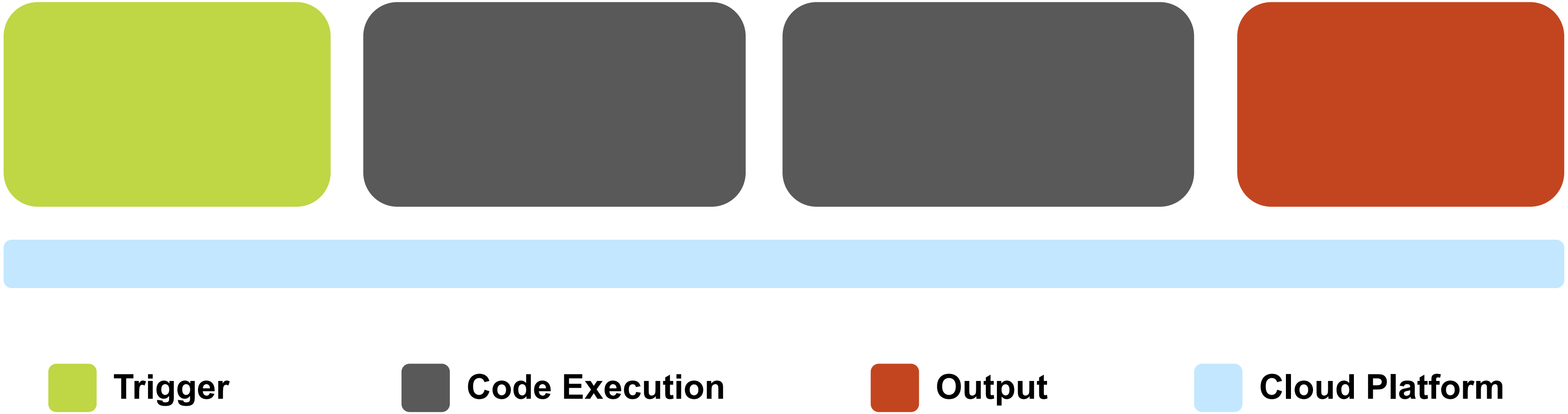
Serverless functions

Server vs Serverless

Items	Server	Serverless
Configuration	Manual	Automatic
Scalability	Fixed	Unlimited
Elasticity	Fixed	Elastic
Billing	Fixed	Pay as you use
Load Balancing	Difficult	Easy

How do serverless functions work?

Serverless functions are a cloud computing service that allows developers to run code without managing infrastructure.



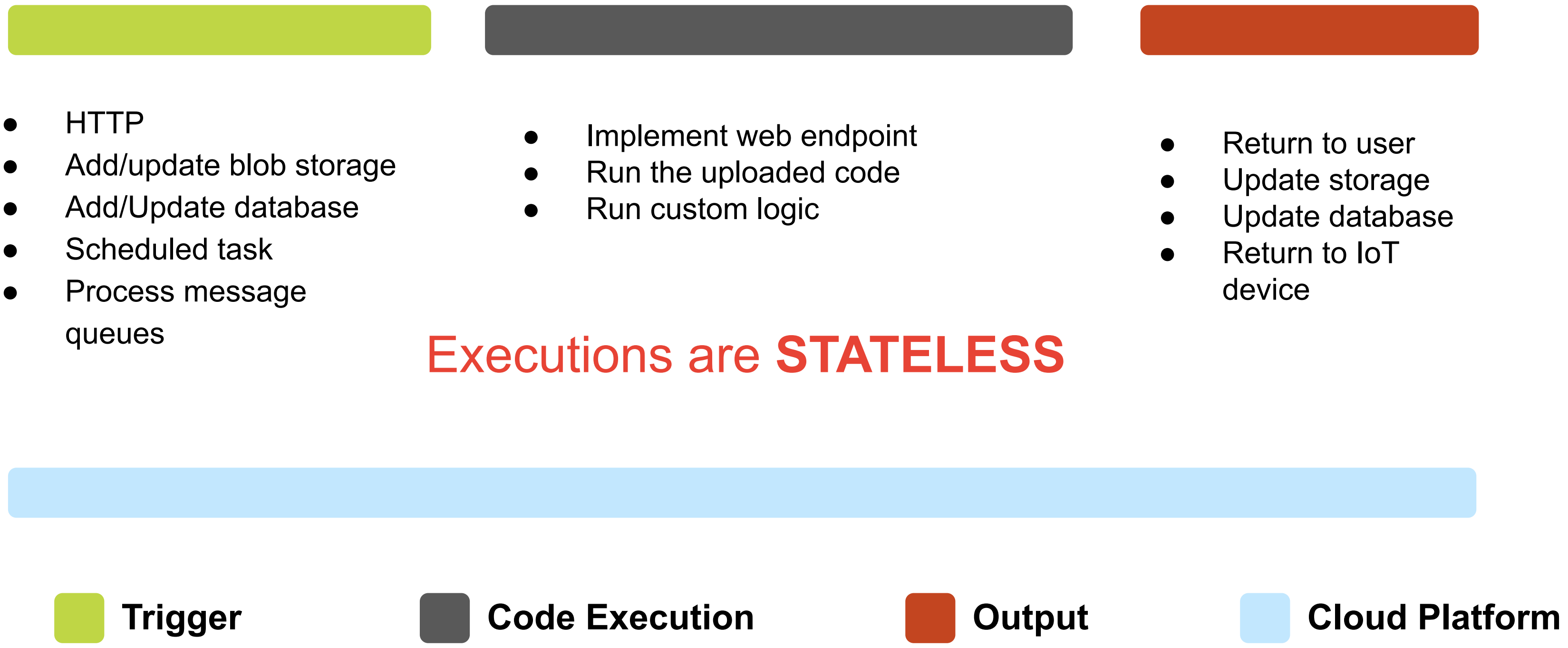
🟩 Trigger

⬛ Code Execution

🟥 Output

🟦 Cloud Platform

What are serverless functions?



- HTTP
- Add/update blob storage
- Add/Update database
- Scheduled task
- Process message queues

- Implement web endpoint
- Run the uploaded code
- Run custom logic

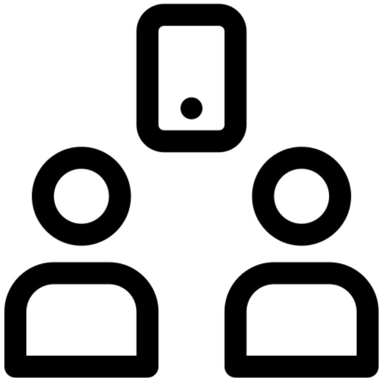
- Return to user
- Update storage
- Update database
- Return to IoT device

Executions are **STATELESS**

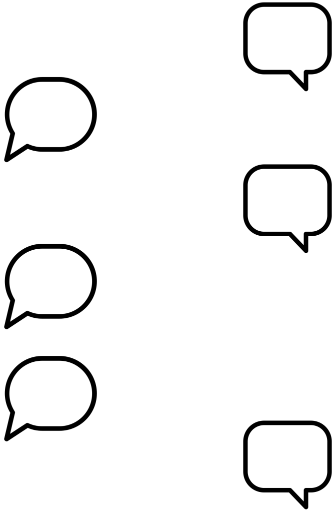
🟩 **Trigger** ⬛ **Code Execution** 🟥 **Output** 🟦 **Cloud Platform**

What is a **STATE ?**

STATE refers to the condition of any given time, or prior knowledge of a task.



Stateful



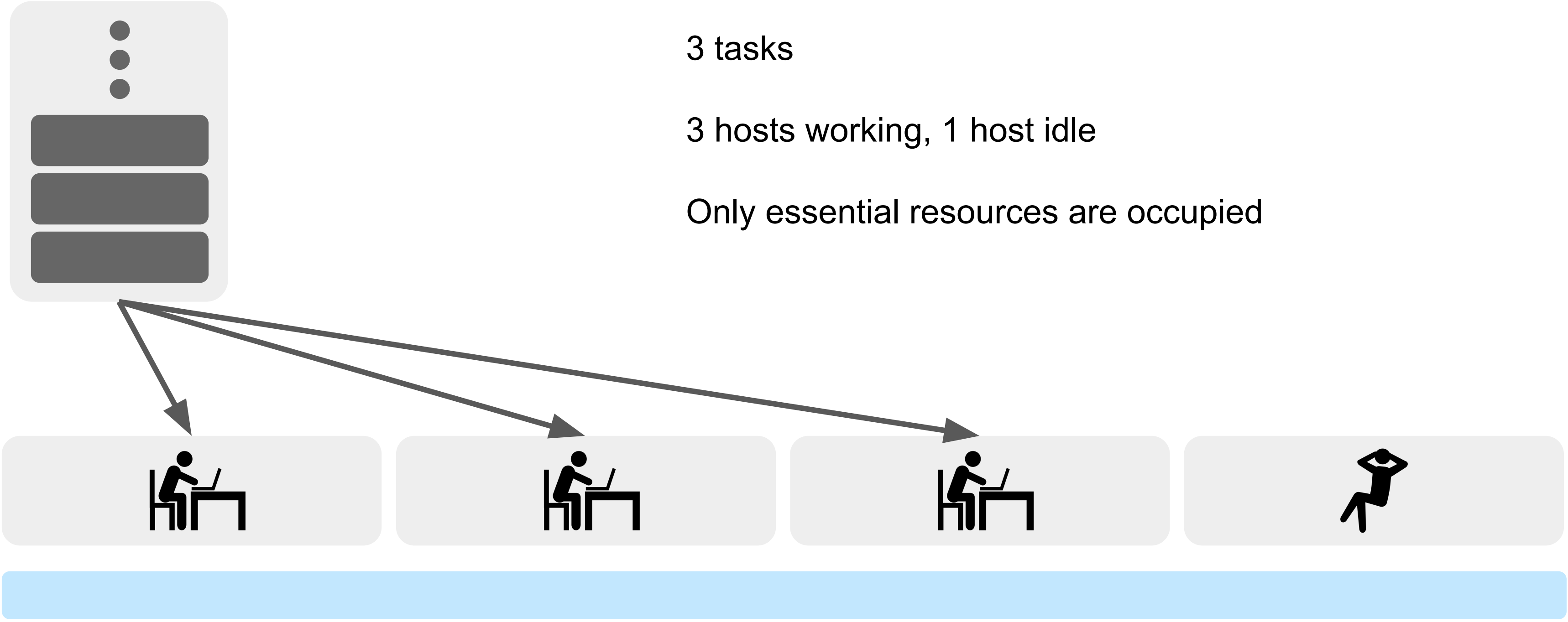
Stateless

How do stateless functions work ?

3 tasks

3 hosts working, 1 host idle

Only essential resources are occupied



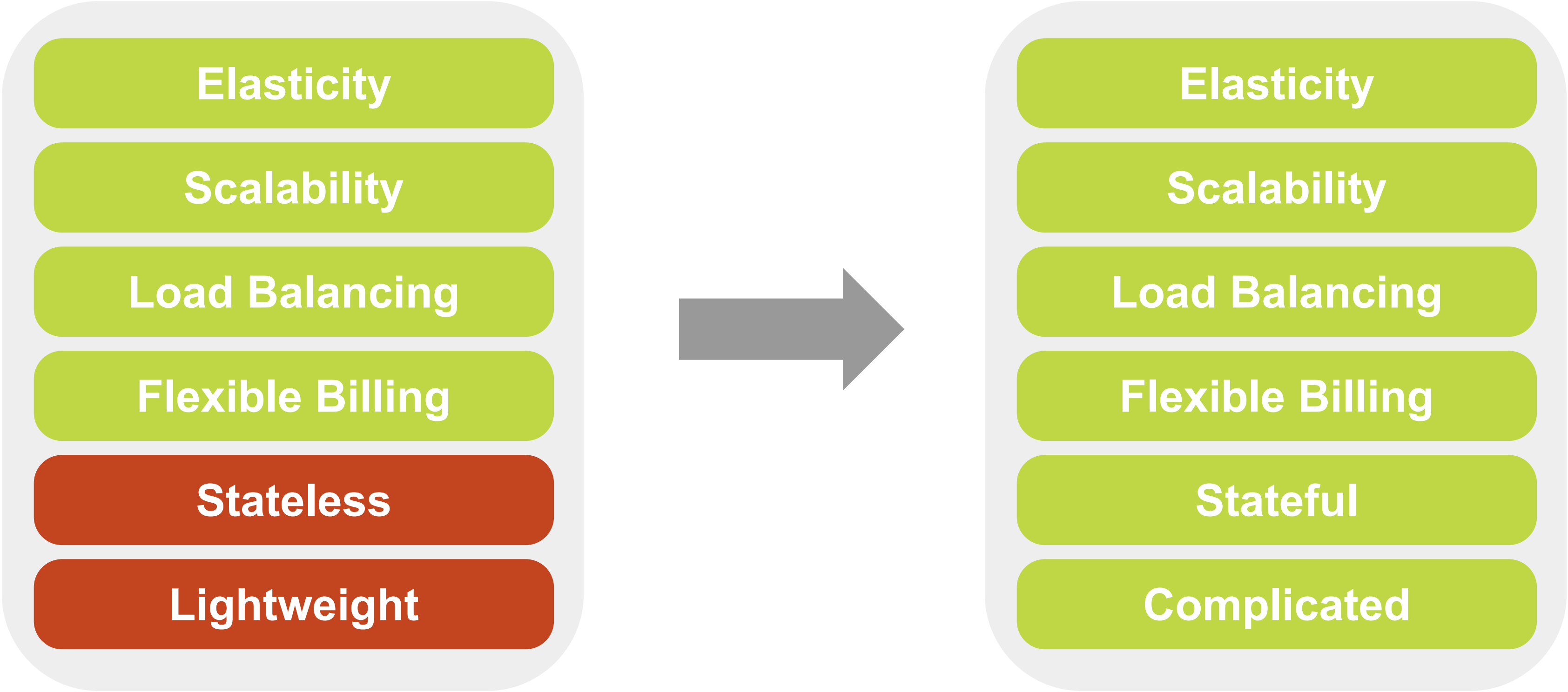
▨ **Host**

■ **Task**

▨ **Cloud Platform**

History background

Stateless HTTP triggers make up less than 50% of invocations



Elasticity

Scalability

Load Balancing

Flexible Billing

Stateless

Lightweight

Elasticity

Scalability

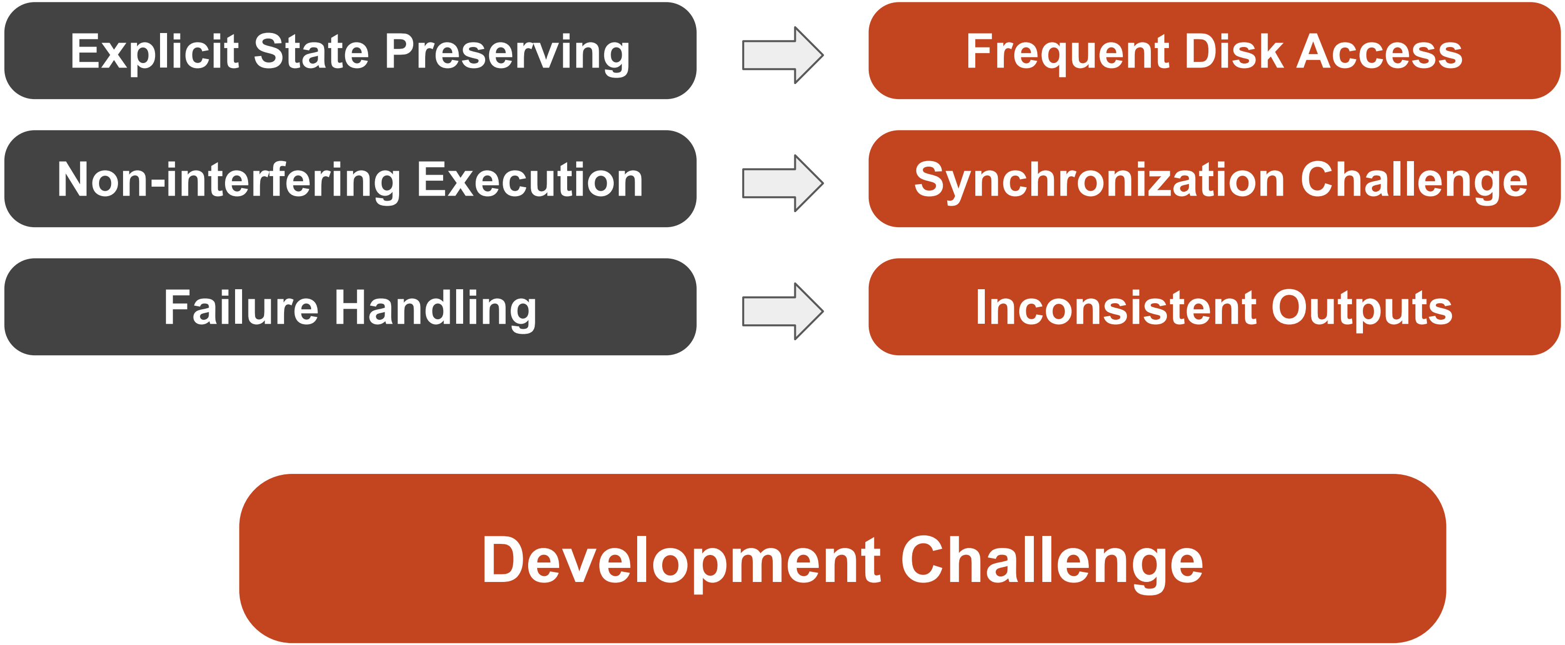
Load Balancing

Flexible Billing

Stateful

Complicated

What could possibly go wrong?



Explicit State Preserving → Frequent Disk Access

Non-interfering Execution → Synchronization Challenge

Failure Handling → Inconsistent Outputs

Development Challenge

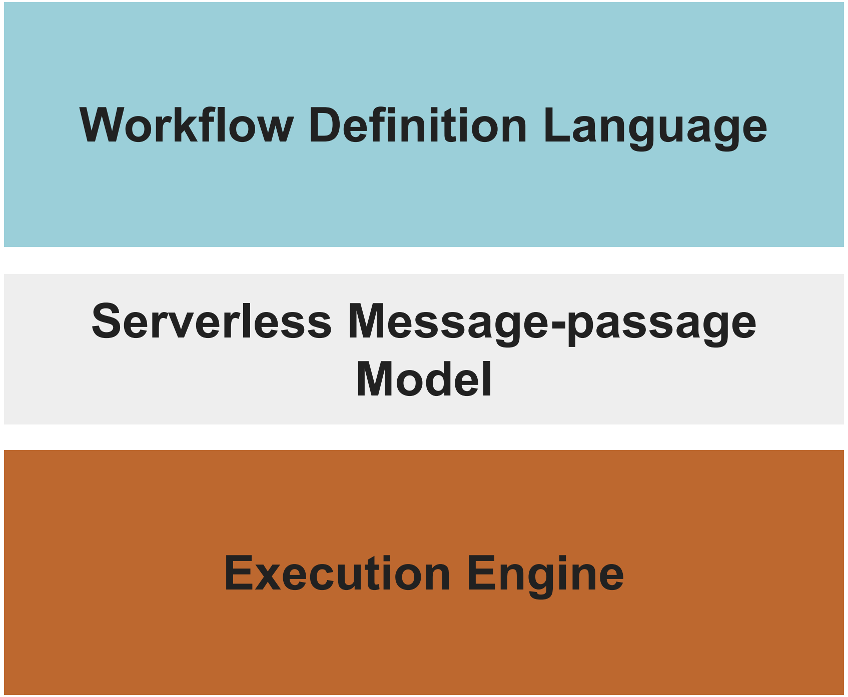
Key motivation

Less Frequent Disk Access

Implicit Synchronizations

Implicit Failure Handling

Stateful Function Structure



Workflow Definition Language

Durable functions - Azure
Step functions - AWS
…

Serverless Message-passage Model

Intermediate layer for decoupling

Execution Engine

Azure storage
Netherite
MS SQL server

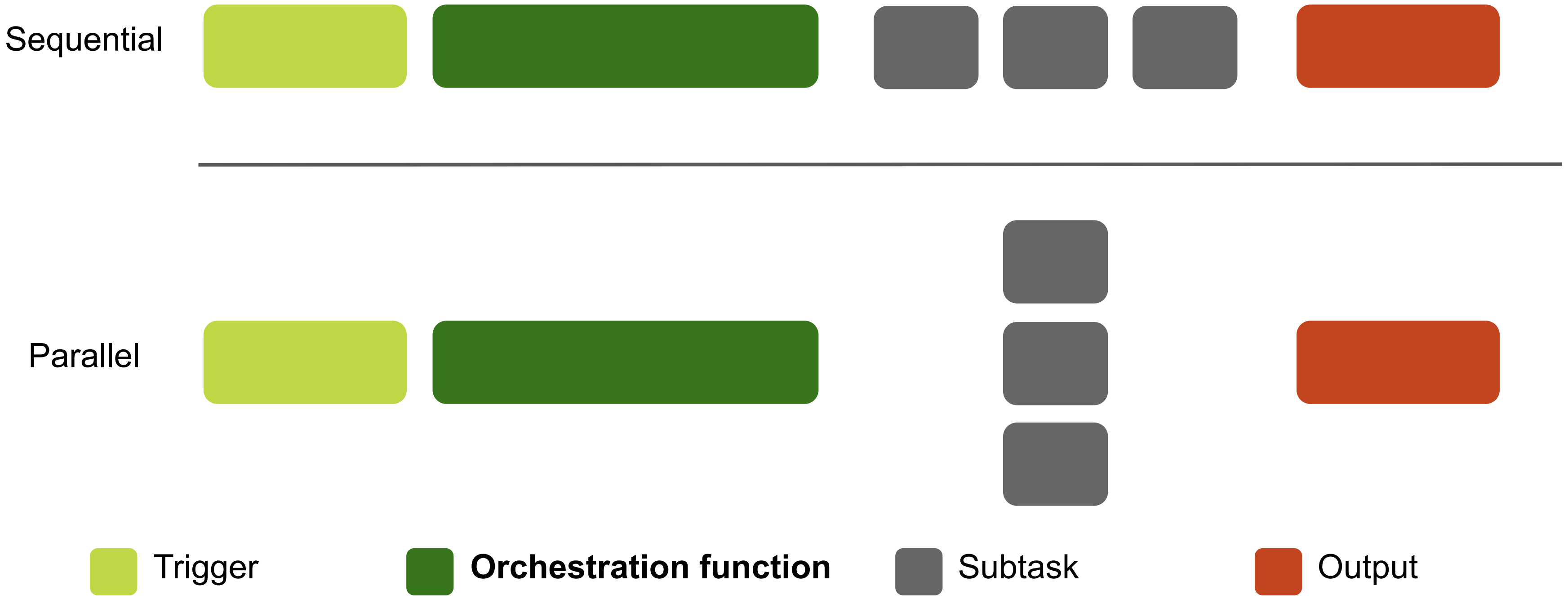
Durable Functions

Orchestration function

Entity function

Critical Section

Orchestration Functions



Sequential

Parallel

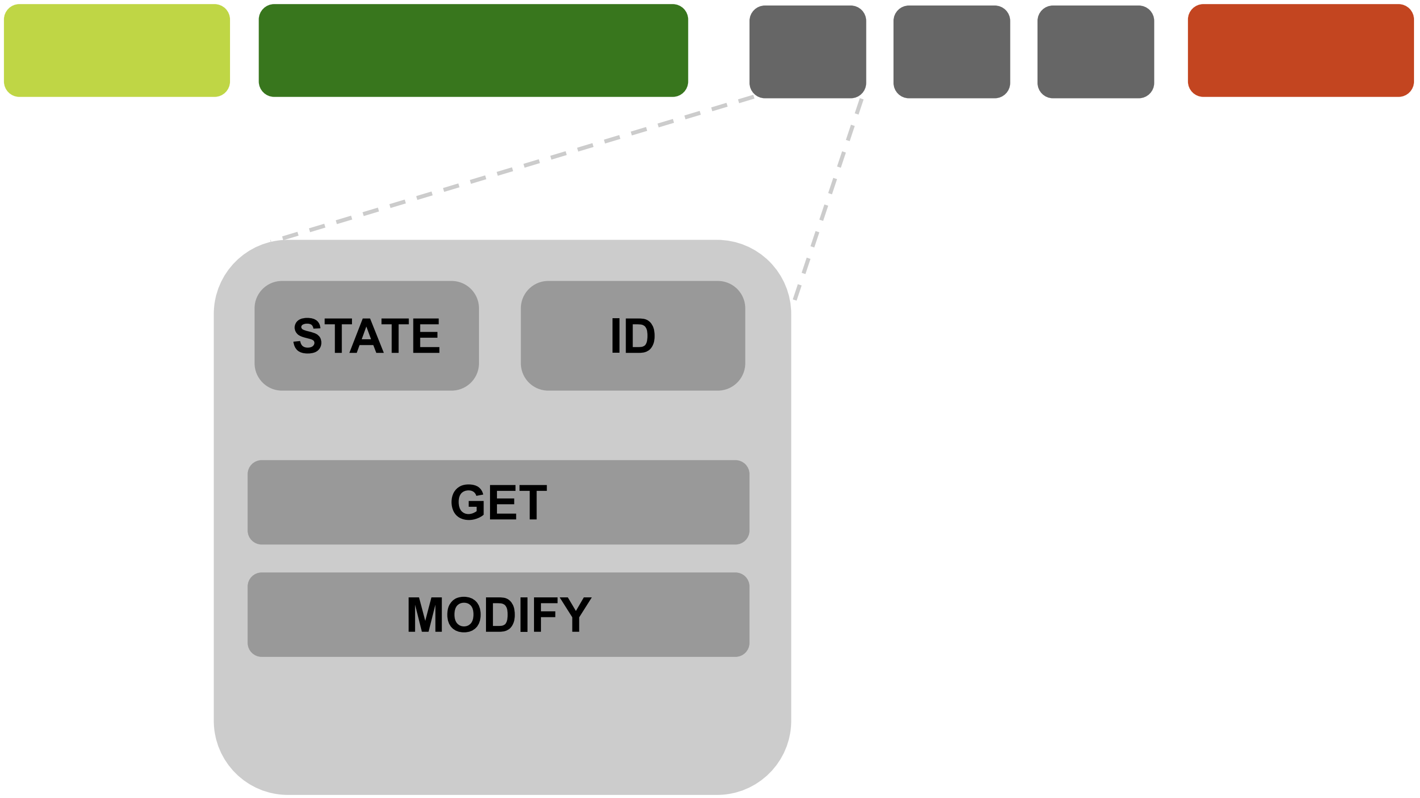
🟩 Trigger

🟩 **Orchestration function**

⬛ Subtask

🟥 Output

Entity



STATE

ID

GET

MODIFY

🟩 Trigger

🟩 Orchestration function

⬛ Subtask

🟥 Output

Critical Sections

Asynchronous locks

🟩 Trigger 🟩 Orchestration function ⬛ Entity ⬚ Critical Section 🟥 Output

Core Problems

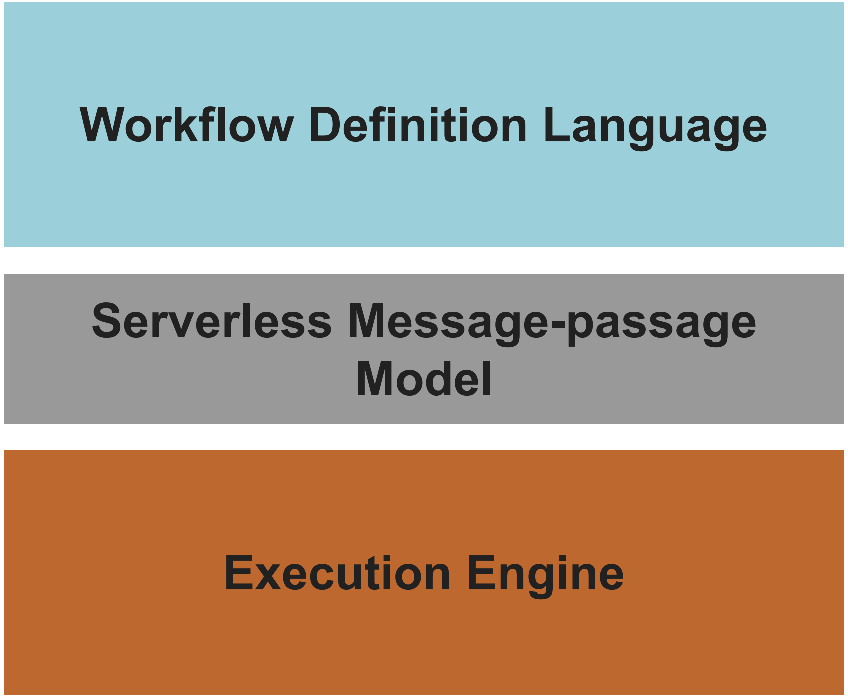
Frequent Disk Access

Synchronization Challenge

Inconsistent Outputs

Message-passing Model

Stateful Function Structure



Workflow Definition Language

Serverless Message-passage Model

Execution Engine

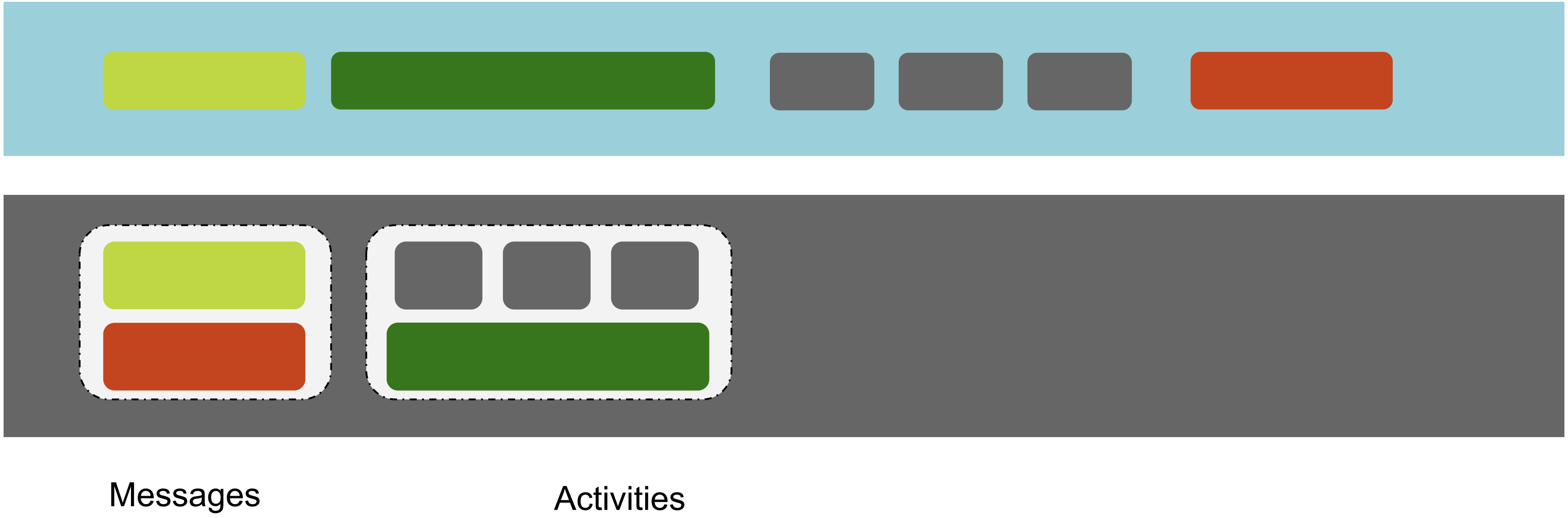
Durable functions - Azure
Step functions - AWS
…

Intermediate layer for decoupling

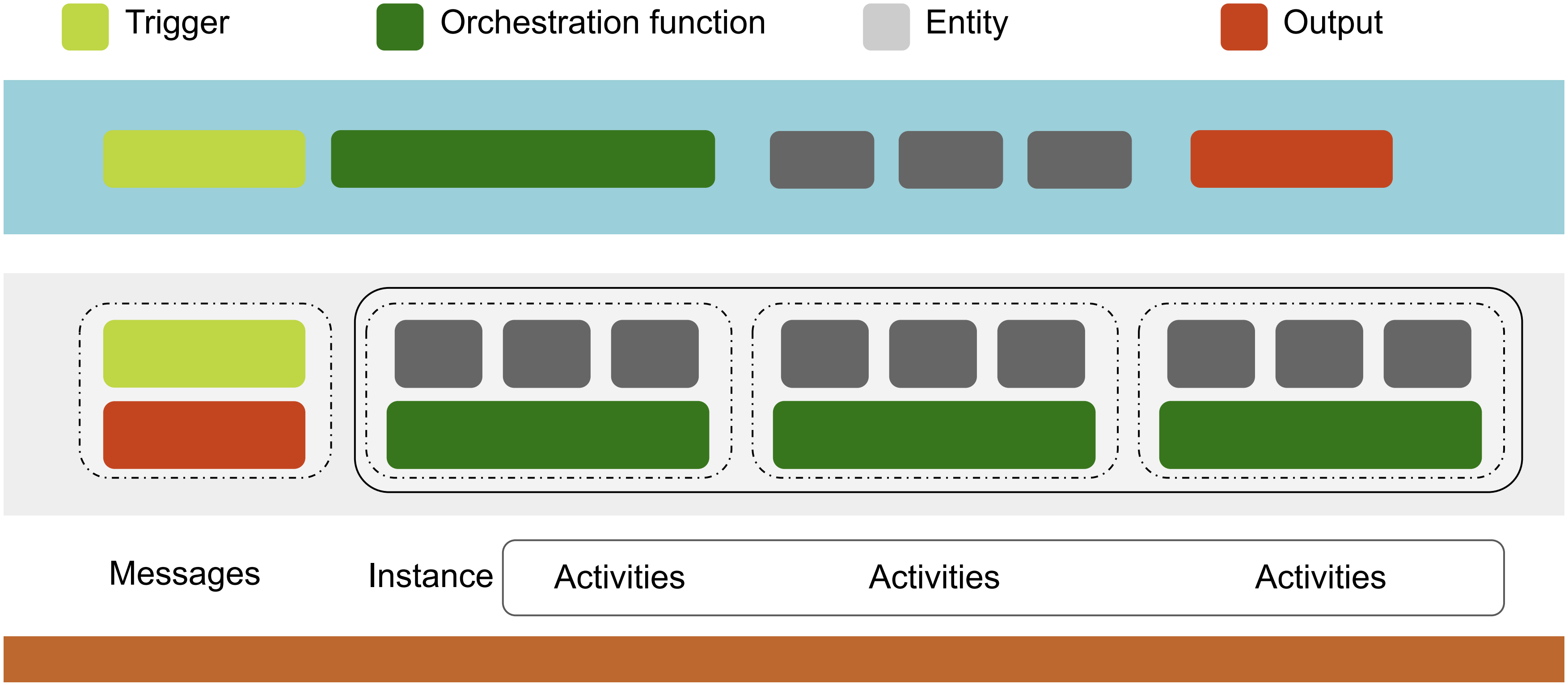
Azure storage
Netherite
MS SQL server

Message-passing Model Architecture

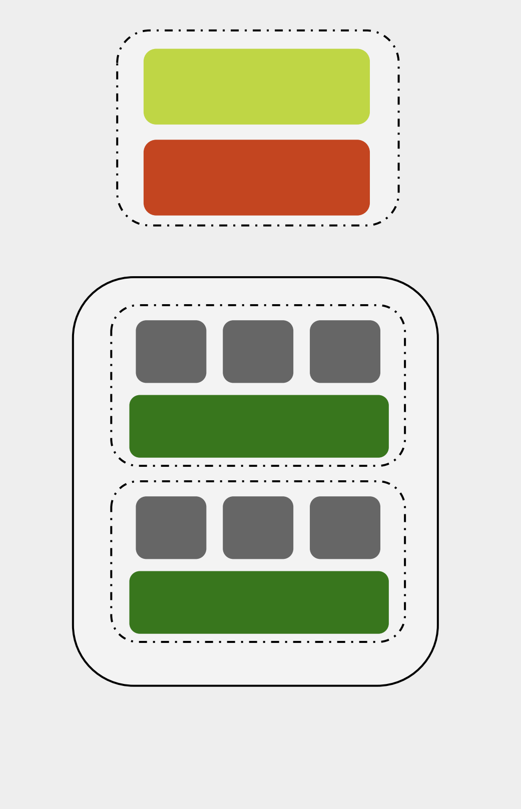
🟩 Trigger 🟩 Orchestration function ⬜ Entity 🟥 Output



Message-passing Model Architecture

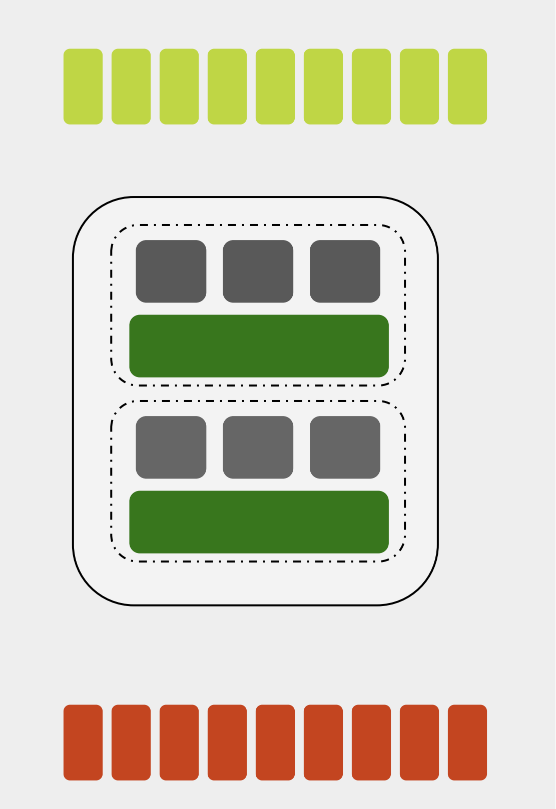


Message-passing Model Implementations



🟥 🟩 Message ⬜ 🟩 Activity □ Instance

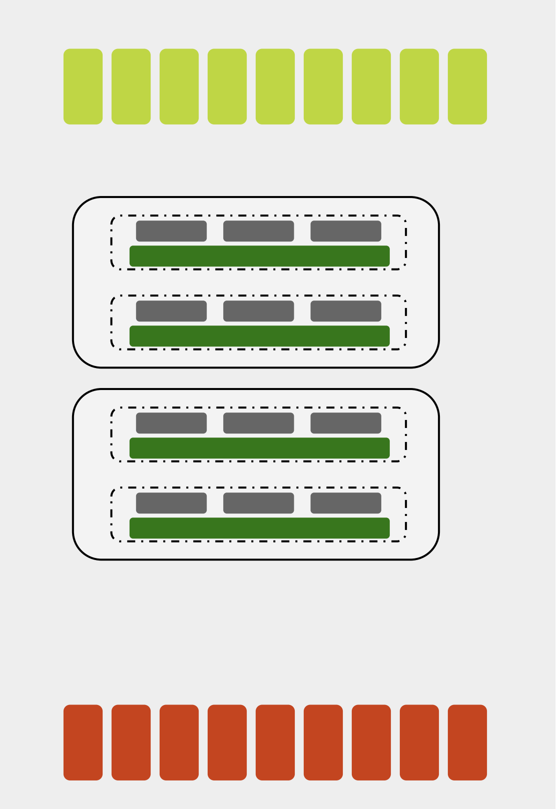
Message-passing Model



Messages (Triggers and outputs) are stacked in queues.

🟥 🟩 Message   ⬜ 🟩 Activity   □ Instance

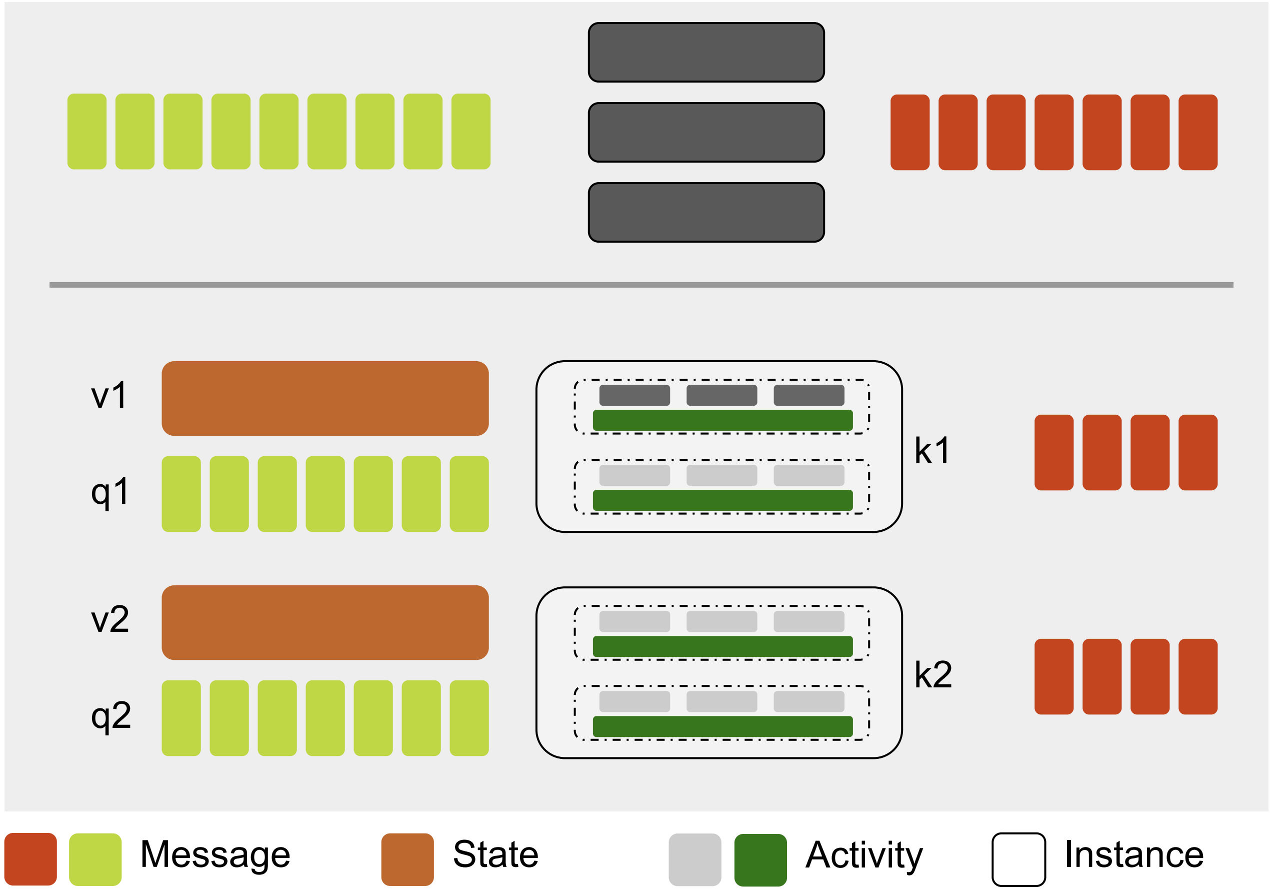
Message-passing Model



Multiple instances are running in parallel

🟥 🟩 Message ⬜ 🟩 Activity □ Instance

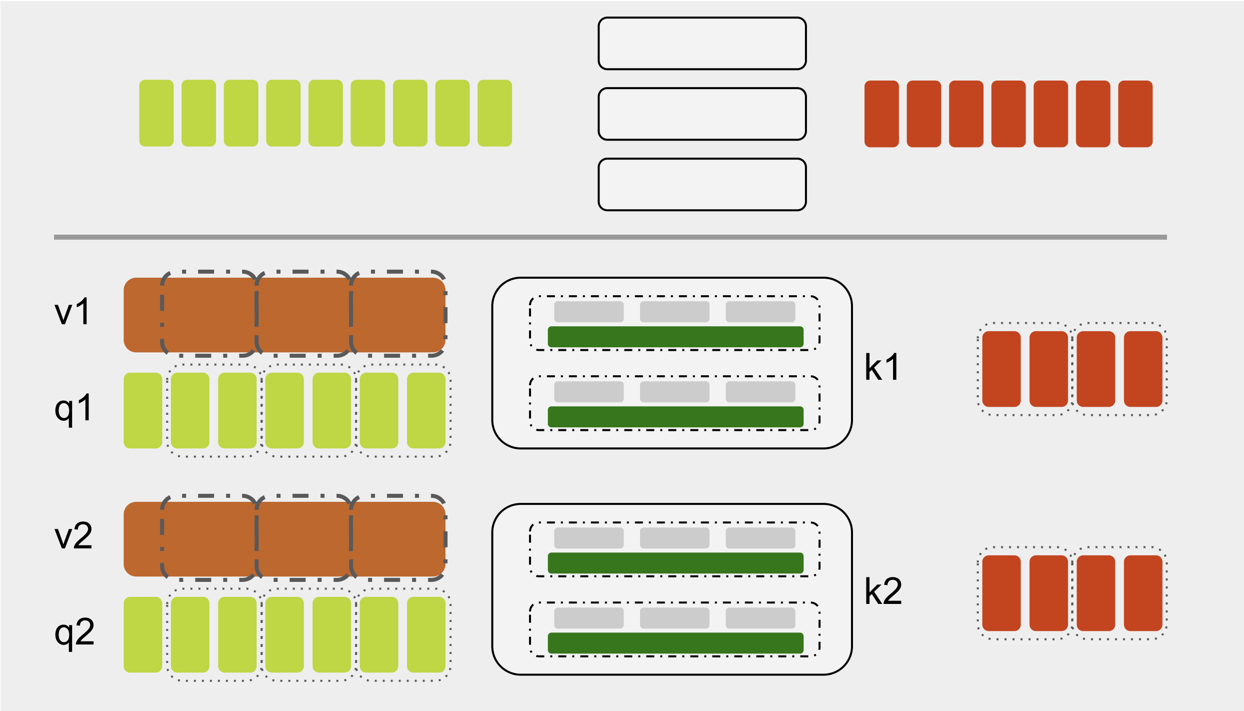
Message-passing Model



Stateless

Stateful

Message-passing Model



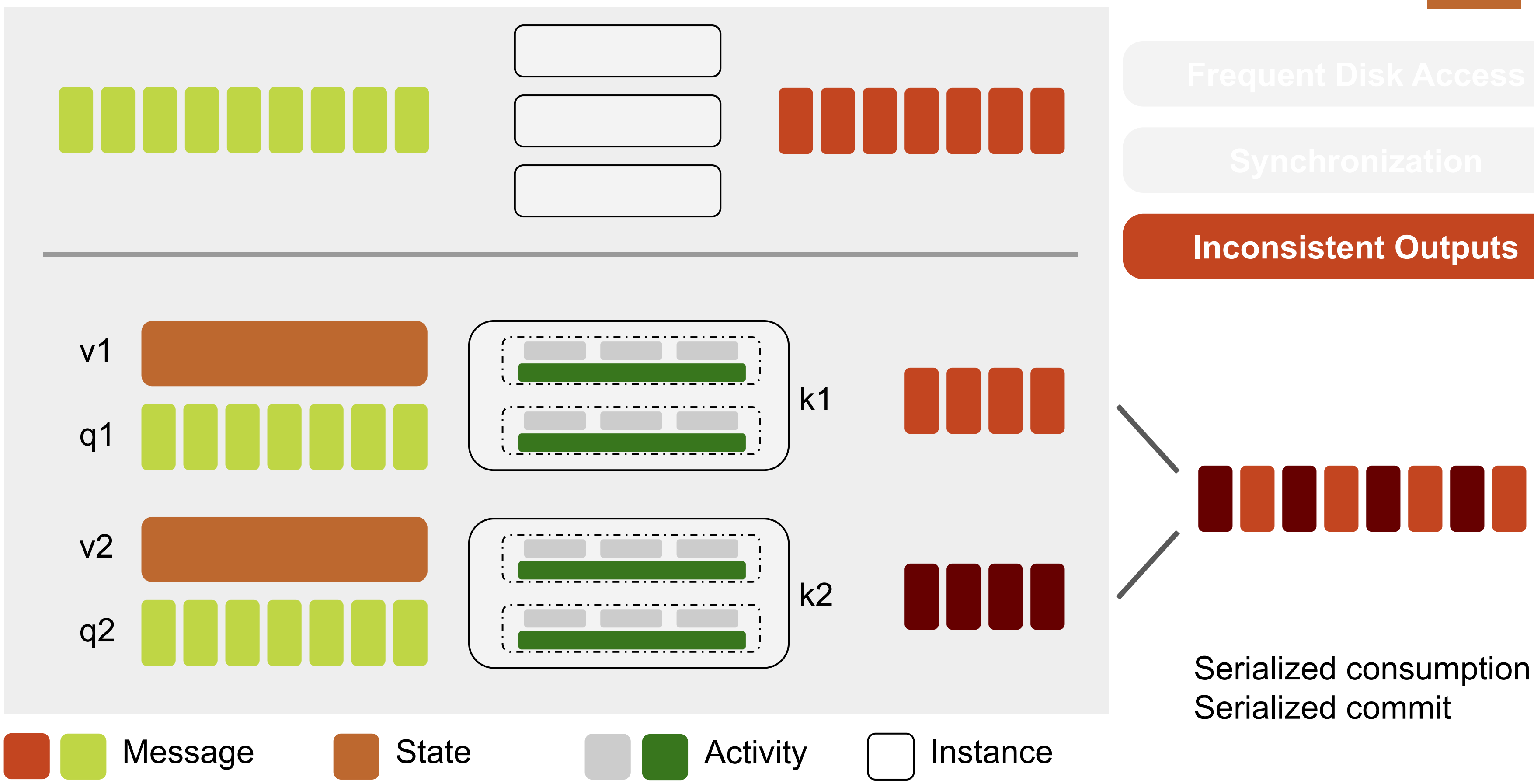
🟥 🟩 Message 🟫 State ⬜ 🟩 Activity □ Instance

Frequent Disk Access

Synchronization

Inconsistent Outputs

Message-passing Model Reliability



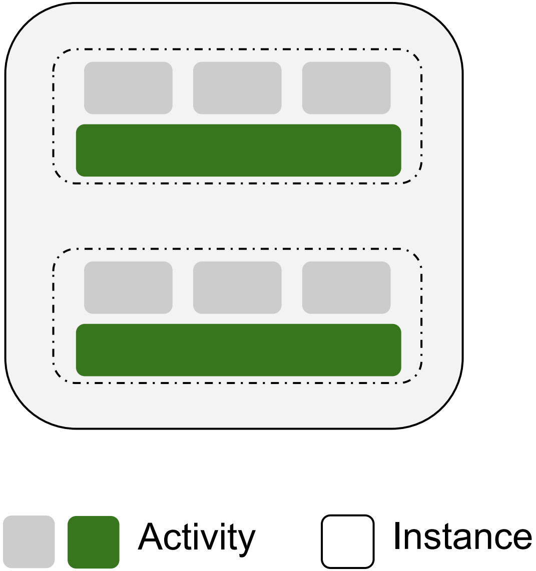
Frequent Disk Access

Synchronization

Inconsistent Outputs

Serialized consumption
Serialized commit

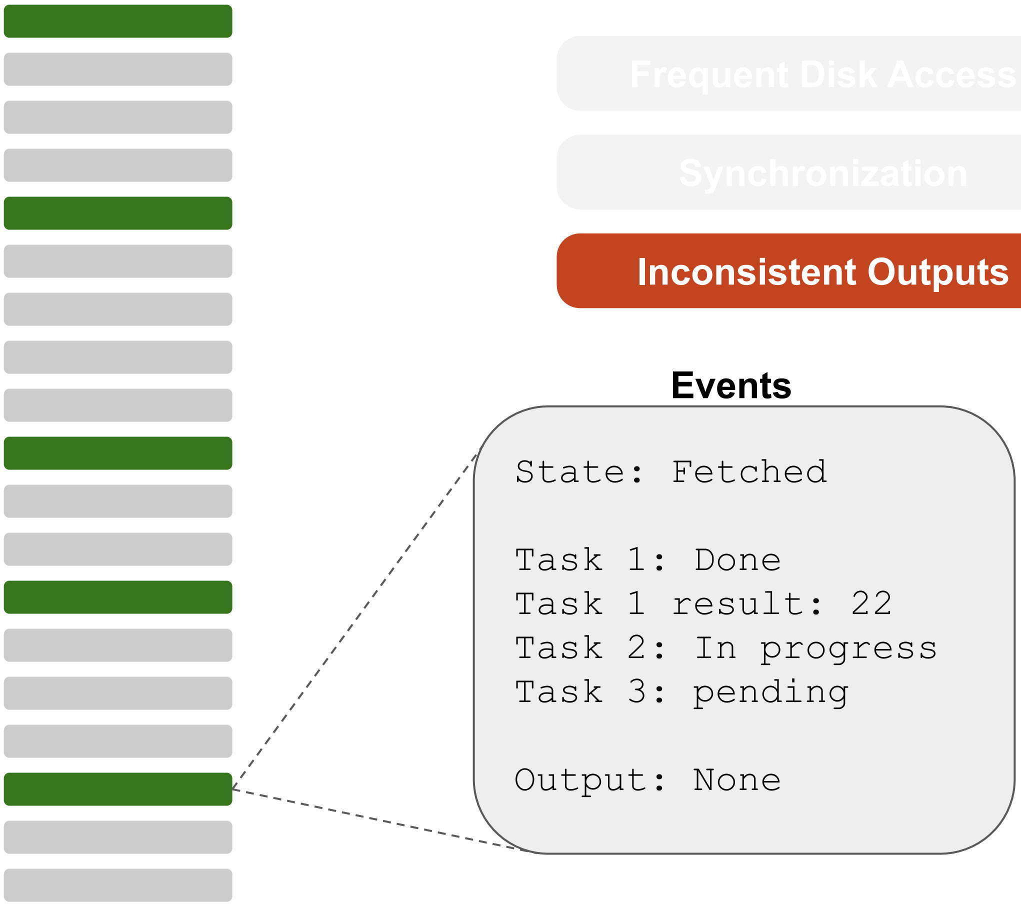
Message-passing Model Reliability



Frequent Disk Access

Synchronization

Inconsistent Outputs



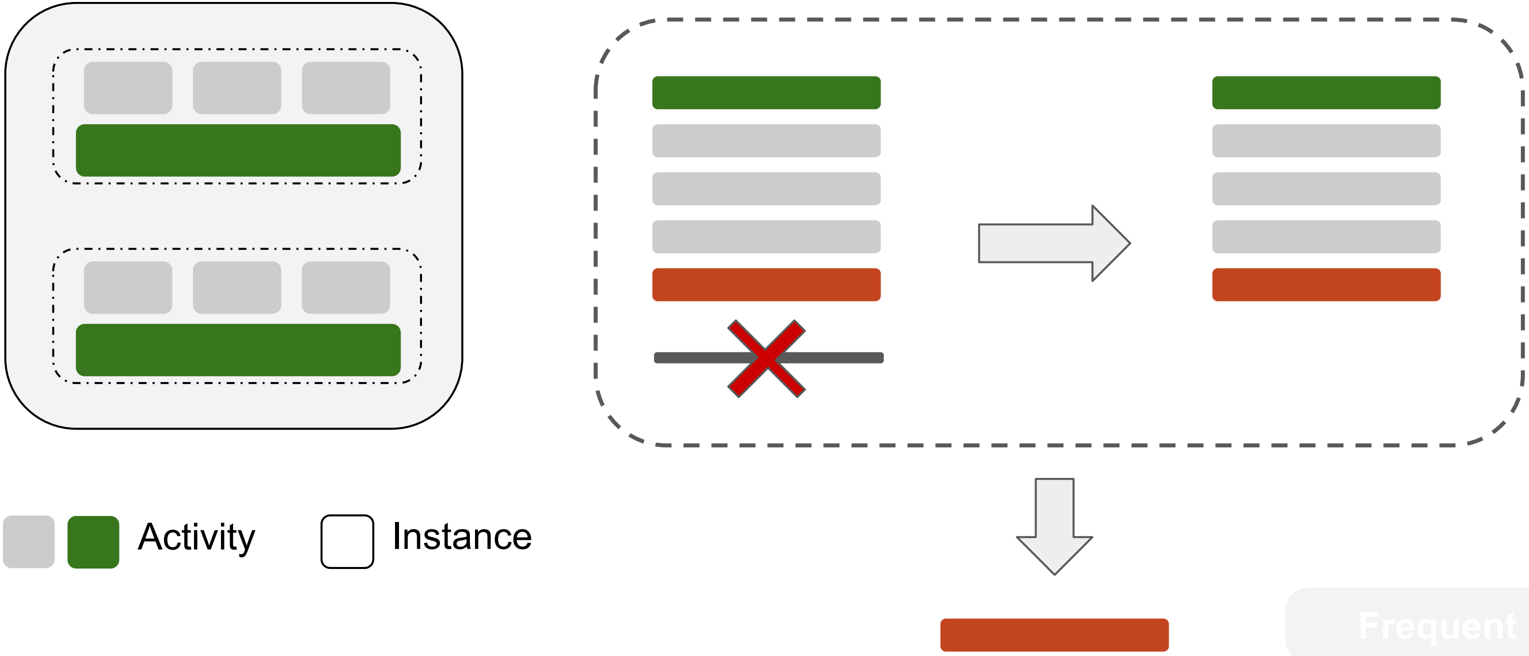
Events

```
State: Fetched

Task 1: Done
Task 1 result: 22
Task 2: In progress
Task 3: pending

Output: None
```

Message-passing Model Reliability



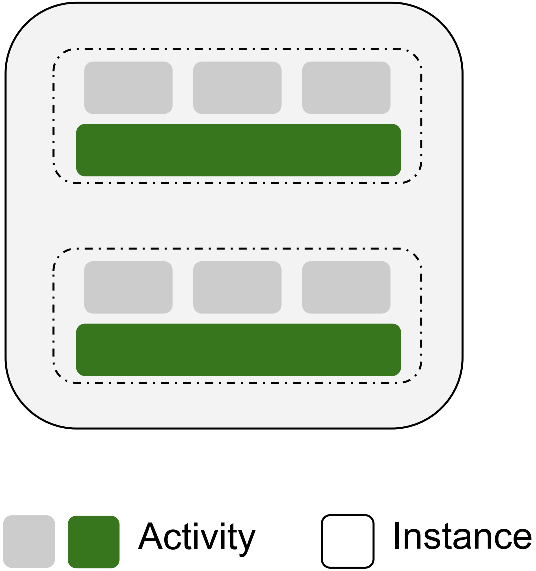
Exactly-once execution

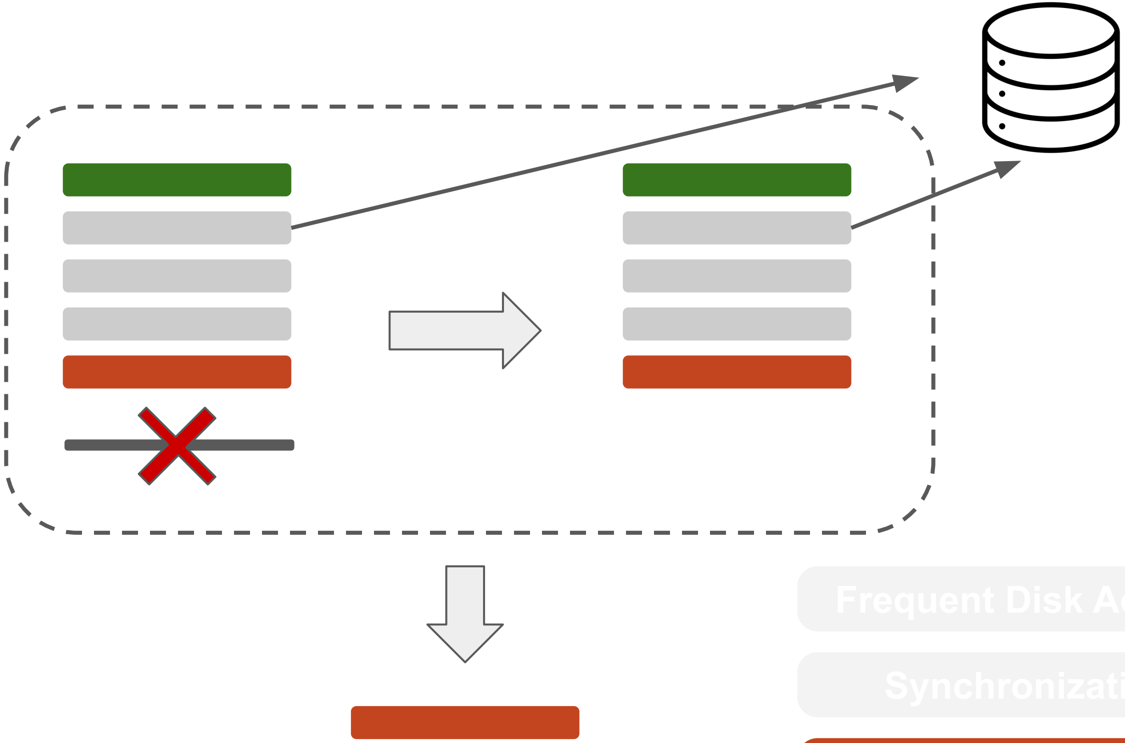
Frequent Disk Access

Synchronization

Inconsistent Outputs

Message-passing Model Reliability





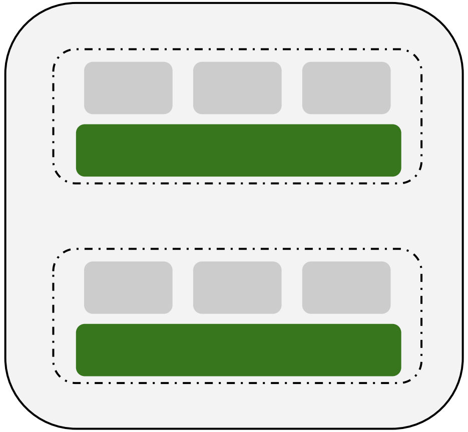
There are external effects!

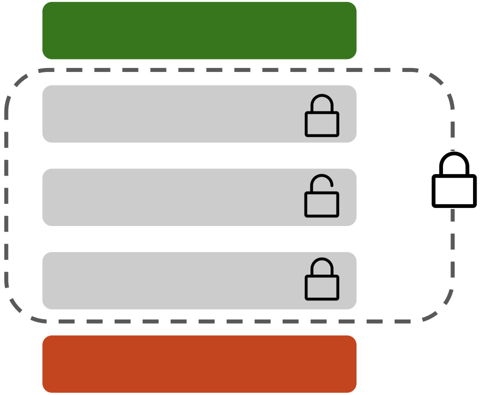
Frequent Disk Access

Synchronization

Inconsistent Outputs

Message-passing Model Reliability





Two-phase locking protocol

Frequent Disk Access

Synchronization

Inconsistent Outputs

▨ 🟩 Activity □ Instance

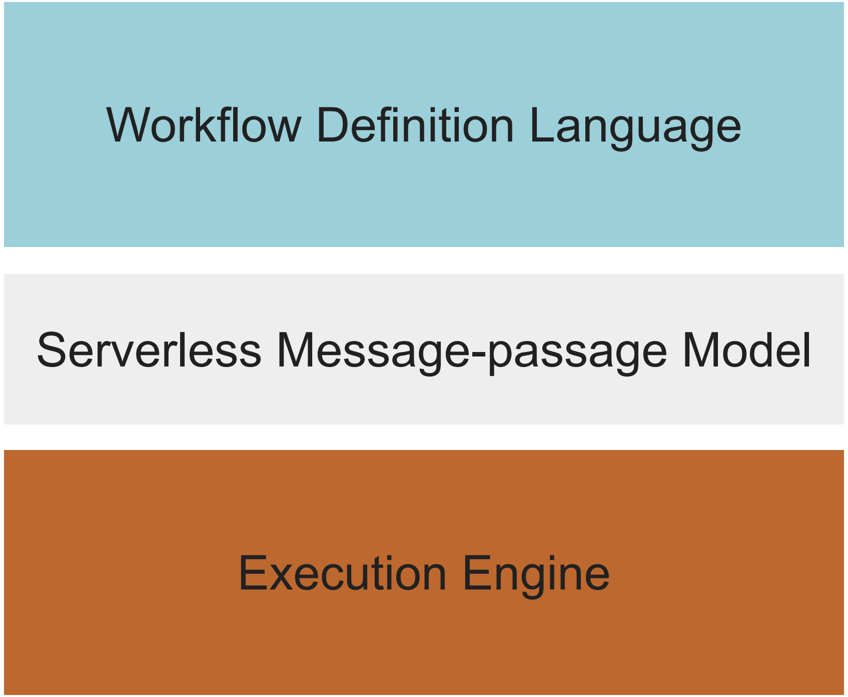
Frequent Disk Access	Batch Operation
Explicit Resource Isolation	Two-phase Lock Protocol
Duplicate/Error Outputs	Serialized Commit Partial History Rollback External Effects

Execution Engines

Original Design

Netherite

Stateful Function Structure



Workflow Definition Language

Durable functions
Step functions
…

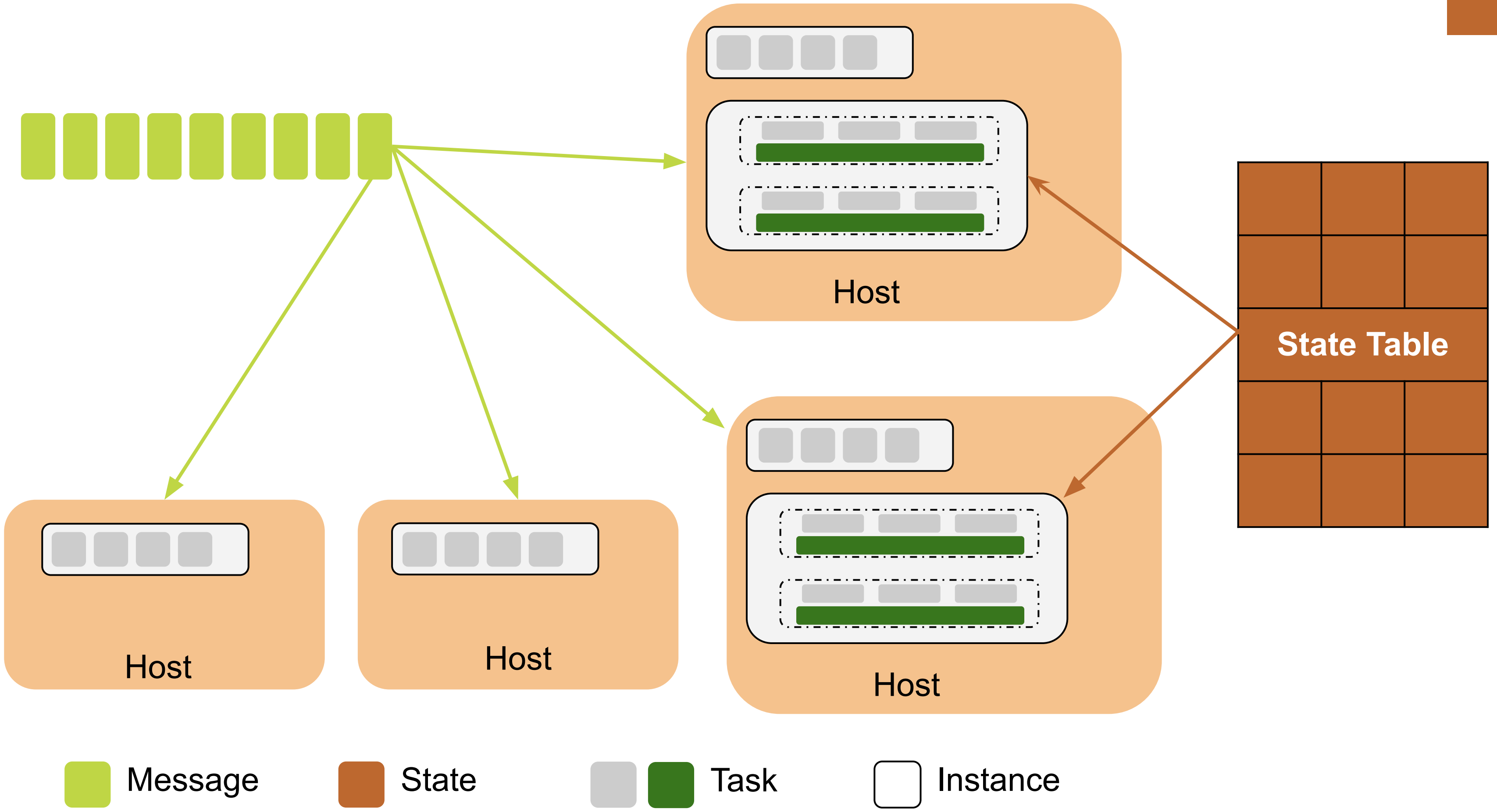
Serverless Message-passage Model

Intermediate layer for decoupling

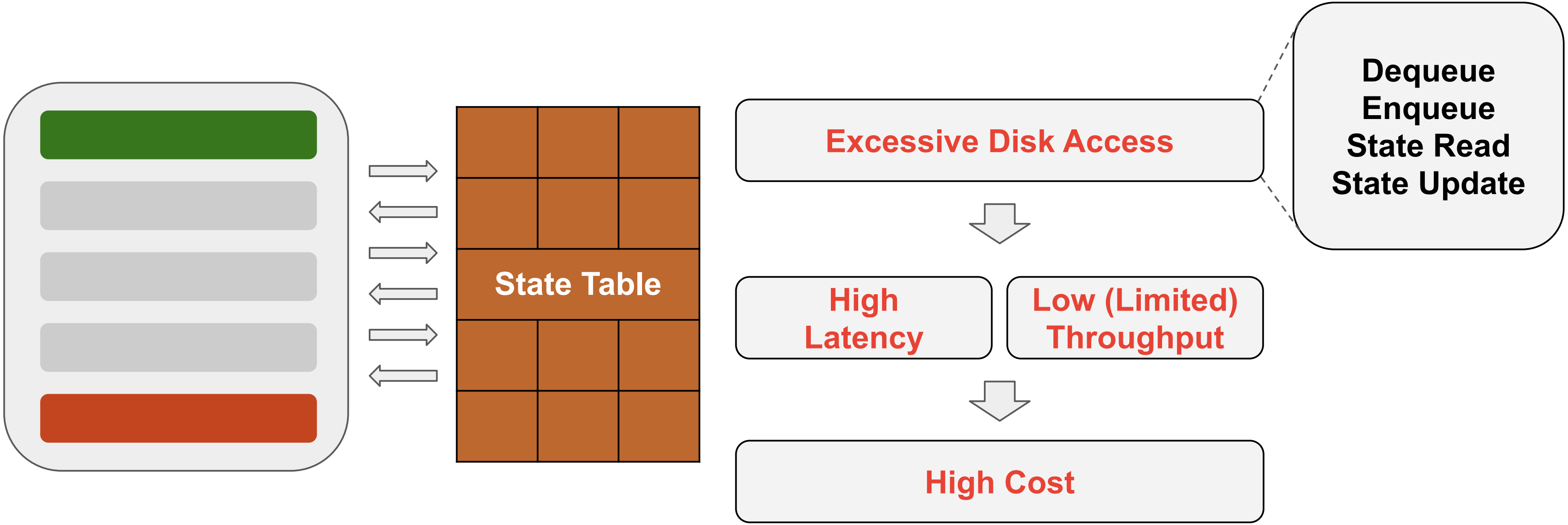
Execution Engine

Azure storage
Netherite
MS SQL server

Original Design Workflow



Performance and Cost



State Table

Excessive Disk Access

Dequeue
Enqueue
State Read
State Update

High Latency

Low (Limited) Throughput

High Cost

🟩 Input ⬜ Subtask 🟥 Output 🟫 State

Core Problems

Frequent Disk Access

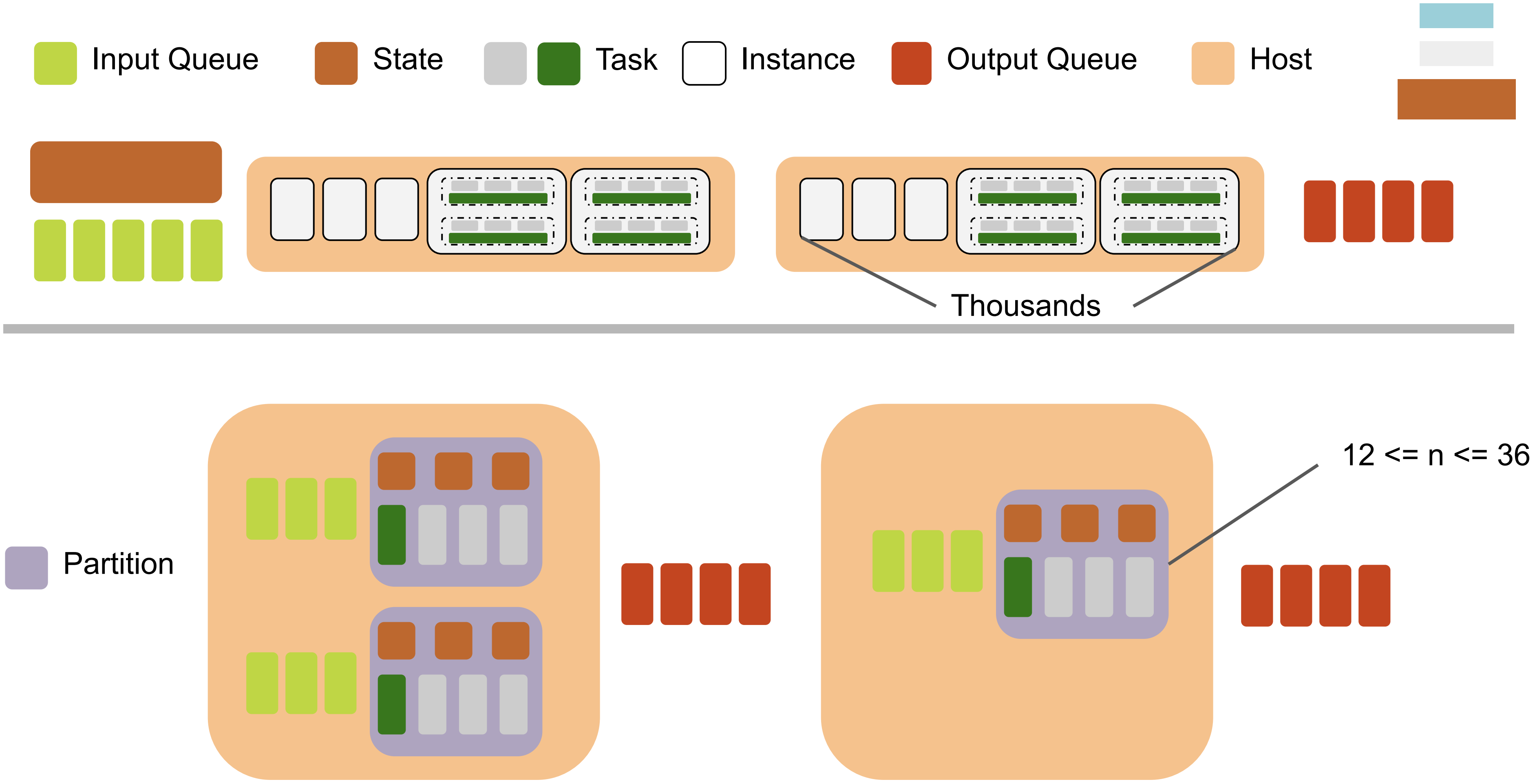
Synchronization Challenge

Inconsistent Outputs

Execution Engines

Original Design

Netherite



Input Queue

State

Task

Instance

Output Queue

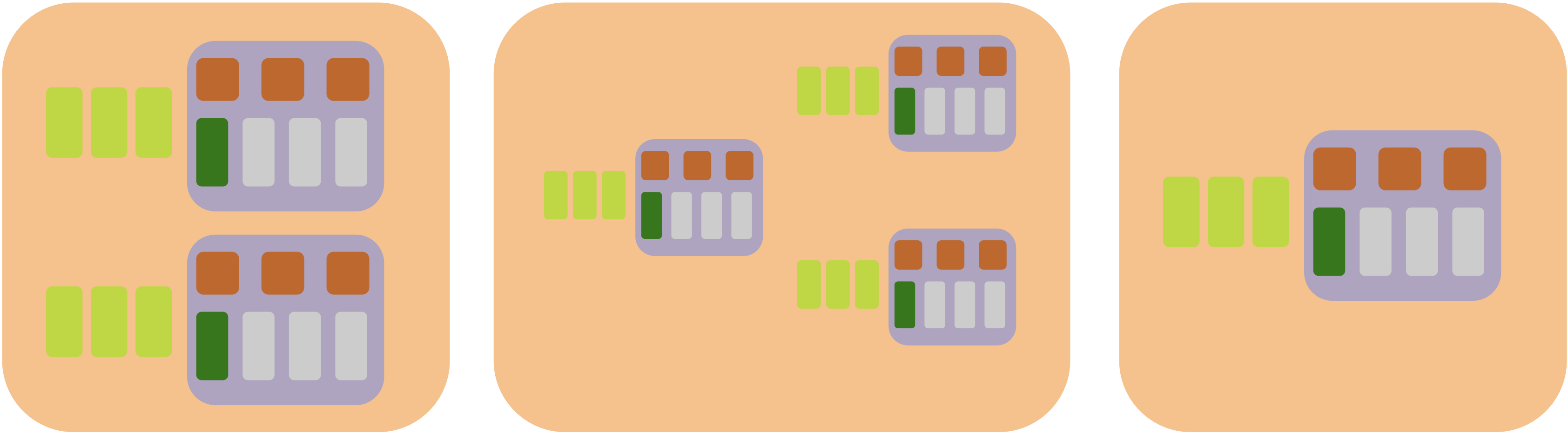
Host

Thousands

Partition

12 <= n <= 36

Netherite - Load Balancing



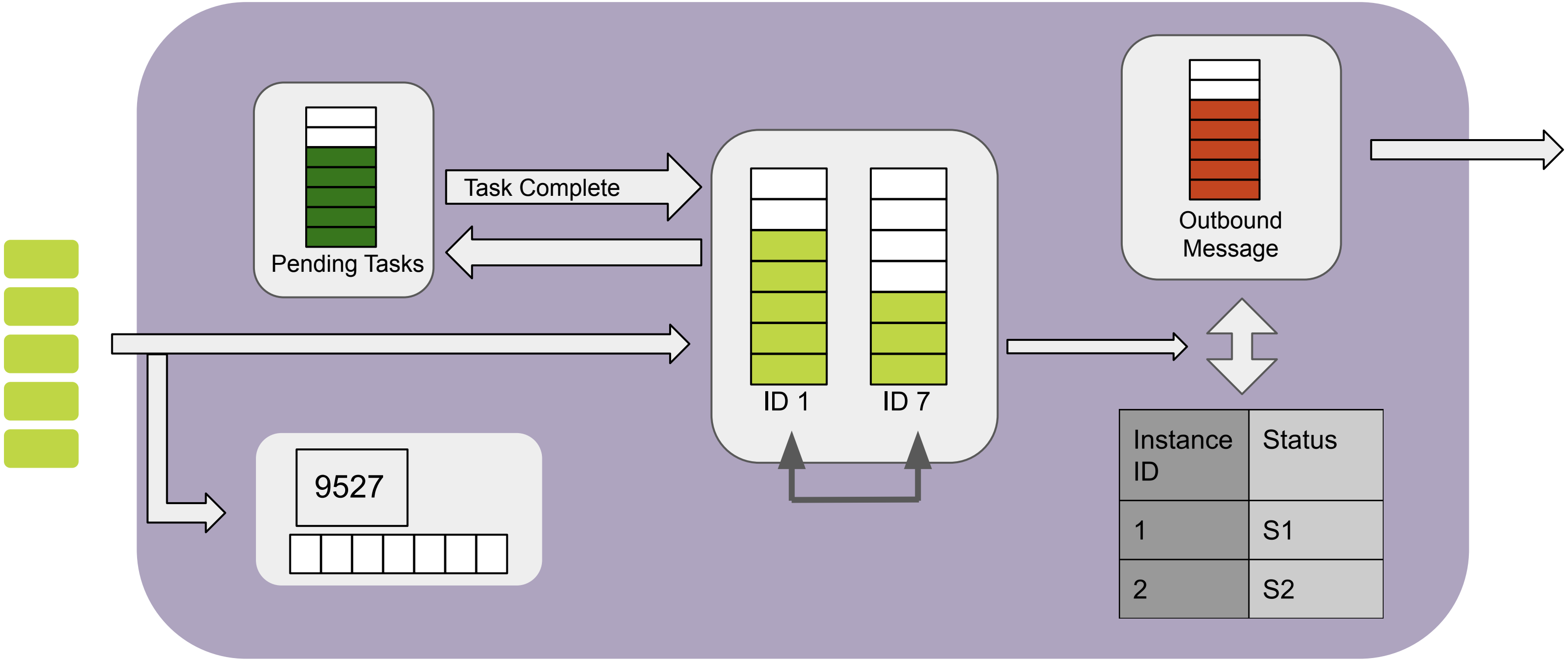
Frequent Disk Access

Synchronization

Inconsistent Outputs

🟩 Message   🟫 State   ⬜ 🟩 Task   🟪 Partition   🟧 Host

Netherite - Workflow



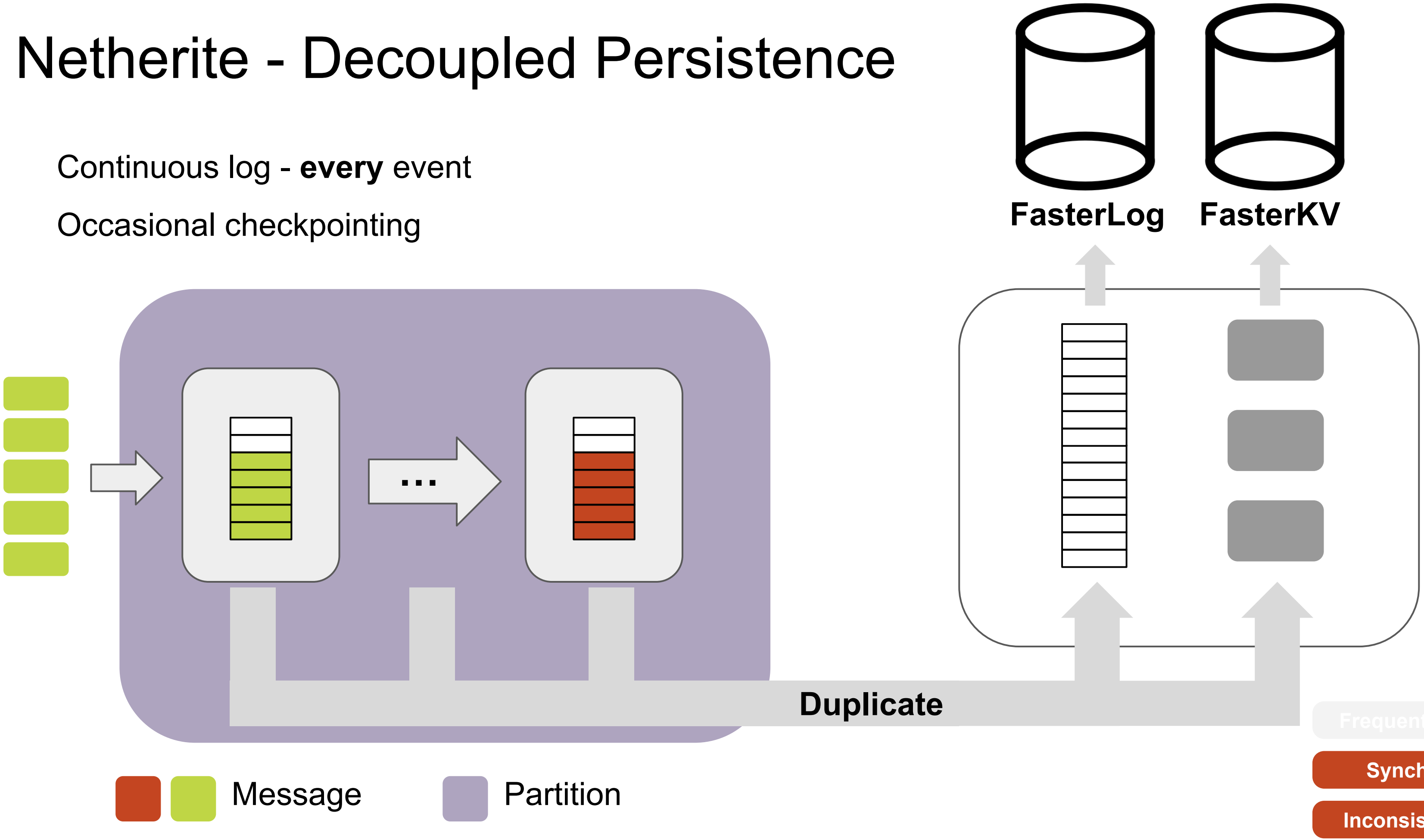
Instance ID	Status
1	S1
2	S2

🟥 🟩 Message 🟧 State 🟩 Task 🟪 Partition

Netherite - Decoupled Persistence

Continuous log - **every** event

Occasional checkpointing



Frequent Disk Access

Synchronization

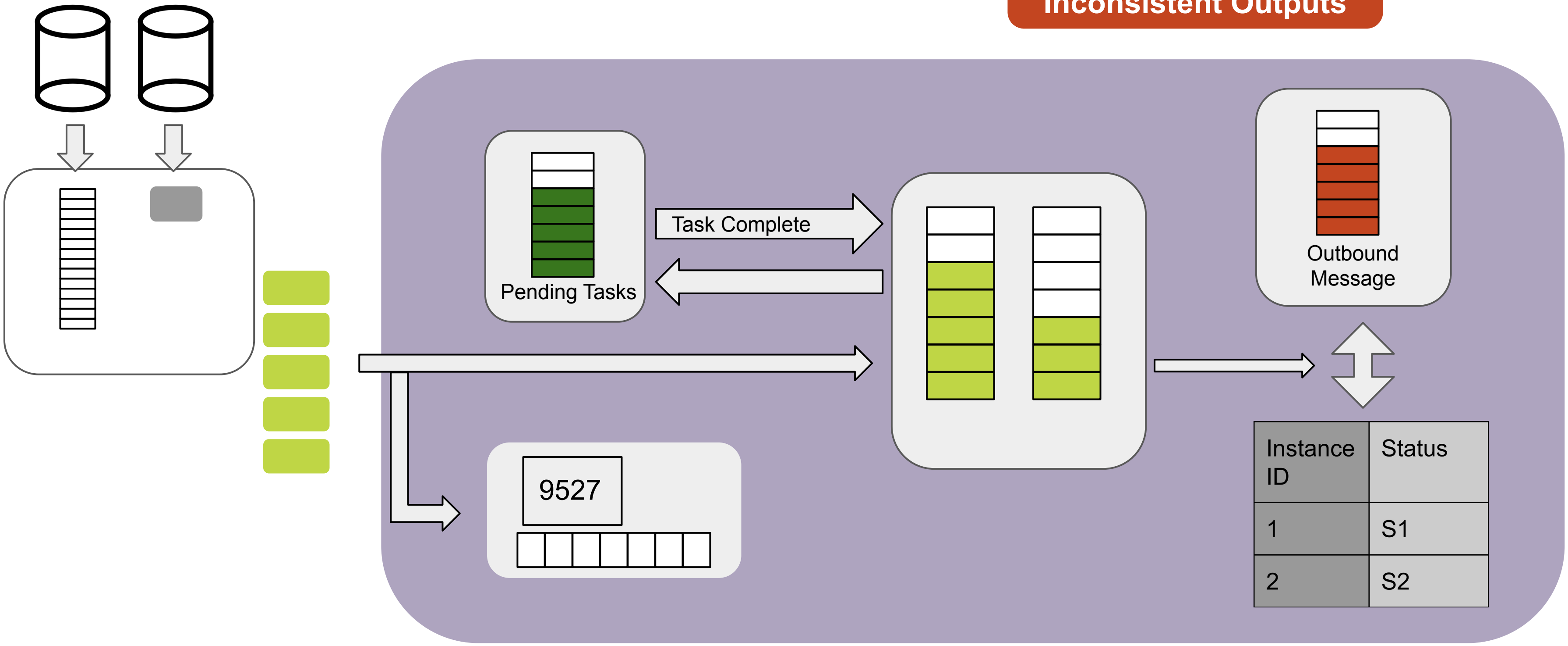
Inconsistent Outputs

Frequent Disk Access

Synchronization

Inconsistent Outputs

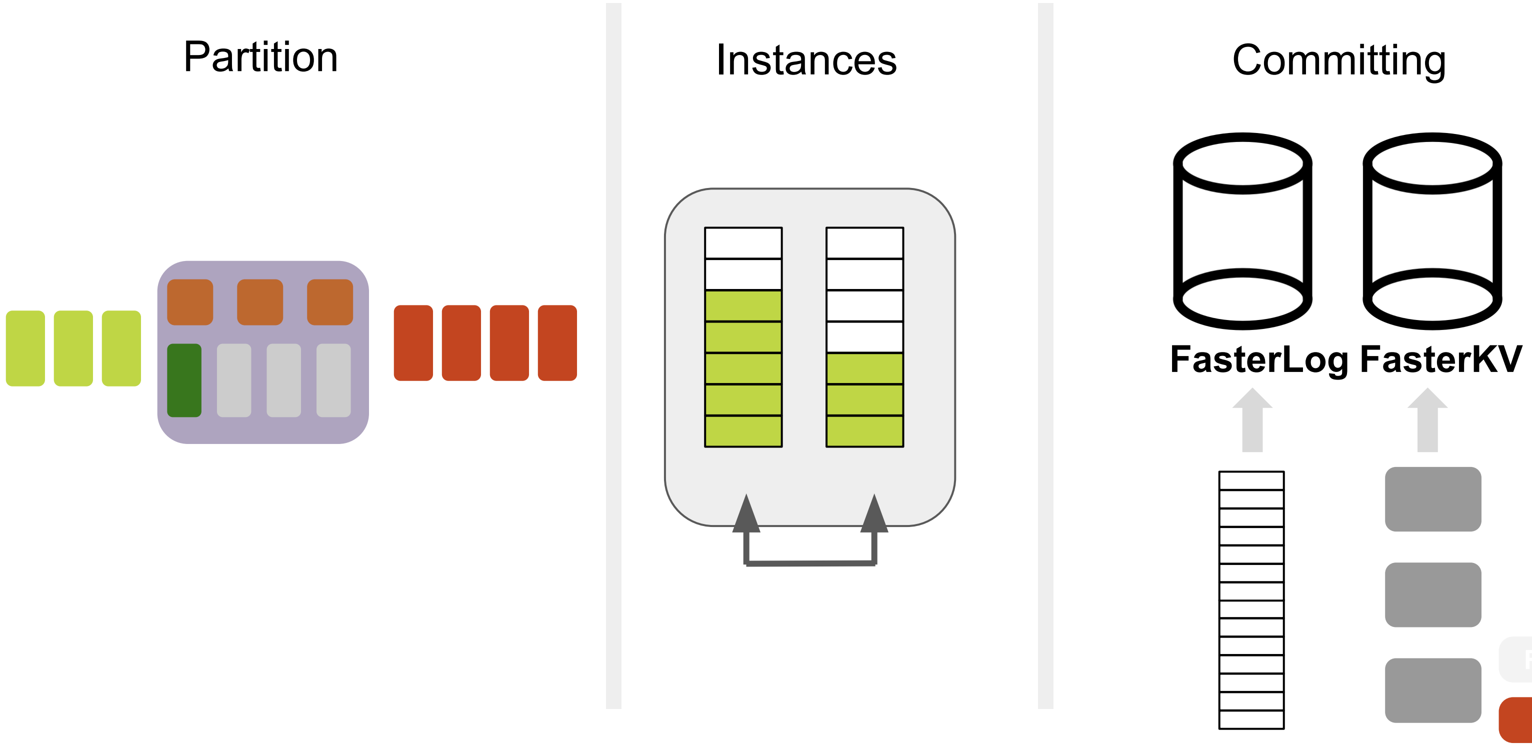
Netherite - Recovery



Instance ID	Status
1	S1
2	S2

🟥 🟩 Message 🟩 Task 🟪 Partition

Partition State Persistence - Serialized Execution



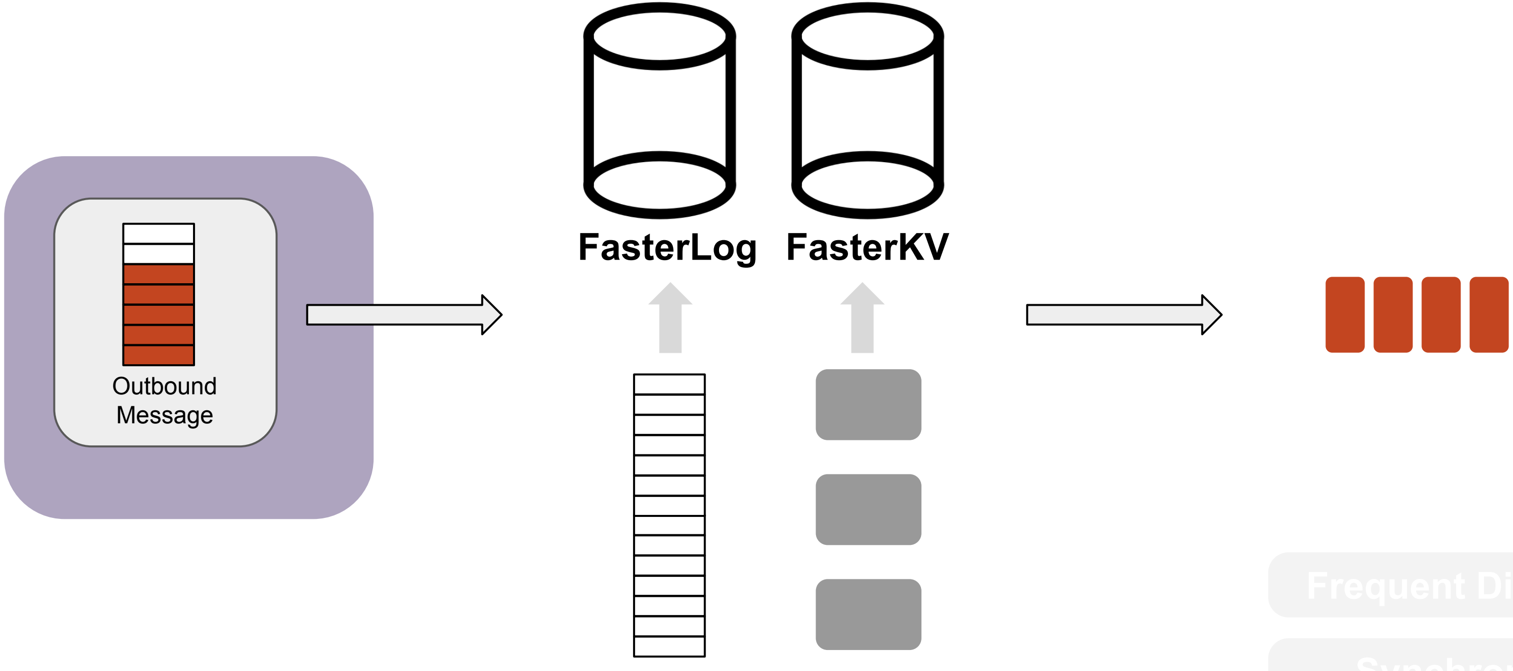
Frequent Disk Access

Synchronization

Inconsistent Outputs

🟥🟩 Message 🟧 State ⬜🟩 Task 🟪 Partition

Partition State Persistence - Guaranteed Execution



🟥 Message 🟪 Partition

Frequent Disk Access

Synchronization

Inconsistent Outputs

Frequent Disk Access	Partition Level Commit Load Balancing
Synchronization Challenge	Serialized Execution
Inconsistent Outputs	Decoupled Persistence Recovery Mechanism Guaranteed Execution

Frequent Disk Access	Batch Operation	Partition Level Commit
		Load Balancing
Synchronization Challenge	Two-phase Lock Protocol	Serialized Execution
Inconsistent Outputs	Serialized Commit	Decoupled Persistence
	Partial History	Recovery Mechanism
	Rollback	Guaranteed Execution
	External Effects	

Evaluation

Evaluation …

Throughput

Storage traffic

Evaluation (continued)

Throughput

Storage traffic

Latency

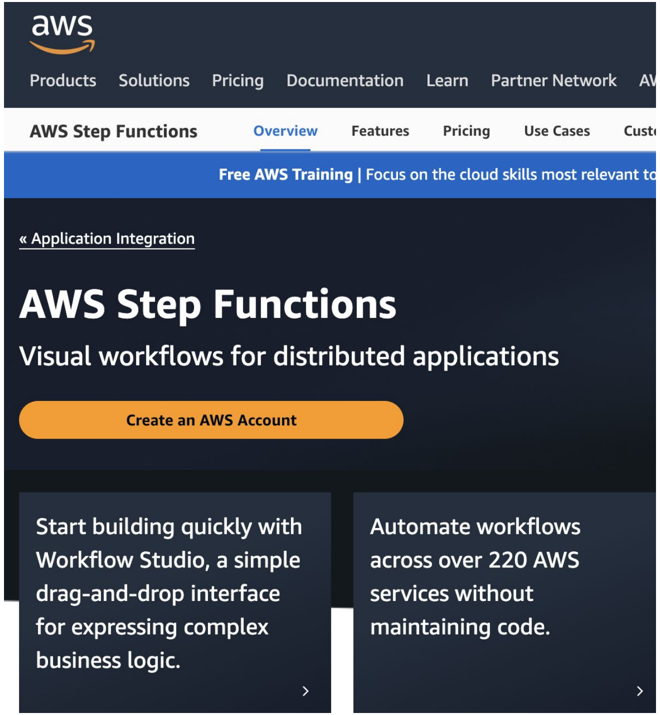
DF alternatives & Latency

What is DF Alternatives?

Durable Function Alternatives

serverless functions with queues or triggers

Durable Function Alternatives



Free AWS Training | Focus on the cloud skills most relevant to

« Application Integration

AWS Step Functions

Visual workflows for distributed applications

Start building quickly with Workflow Studio, a simple drag-and-drop interface for expressing complex business logic.

Automate workflows across over 220 AWS services without maintaining code.

a workflow service

Use JSON

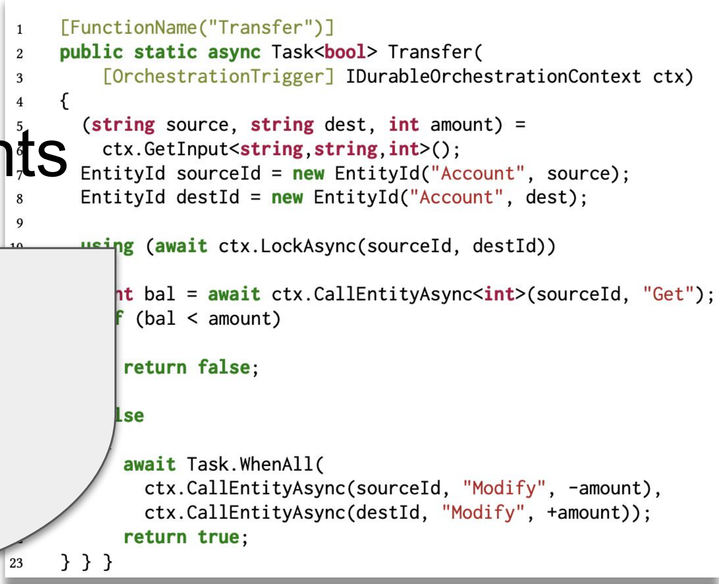
EVALUATION: Throughput Experiments

Hello5
Calls five talks in sequence

Bank
Reliable transfers between accounts

WordCount
Analyzes word frequency

CollisionSearch
Divide-and-conquer searches hash collisions



```
[FunctionName("Transfer")]
public static async Task<bool> Transfer(
    [OrchestrationTrigger] IDurableOrchestrationContext ctx)
{
  (string source, string dest, int amount) =
    ctx.GetInput<string,string,int>();
  EntityId sourceId = new EntityId("Account", source);
  EntityId destId = new EntityId("Account", dest);

  using (await ctx.LockAsync(sourceId, destId))

      nt bal = await ctx.CallEntityAsync<int>(sourceId, "Get");
      f (bal < amount)

        return false;

      lse

        await Task.WhenAll(
          ctx.CallEntityAsync(sourceId, "Modify", -amount),
          ctx.CallEntityAsync(destId, "Modify", +amount));
        return true;
} } }
```

EVALUATION: Methodology

Same Deployment

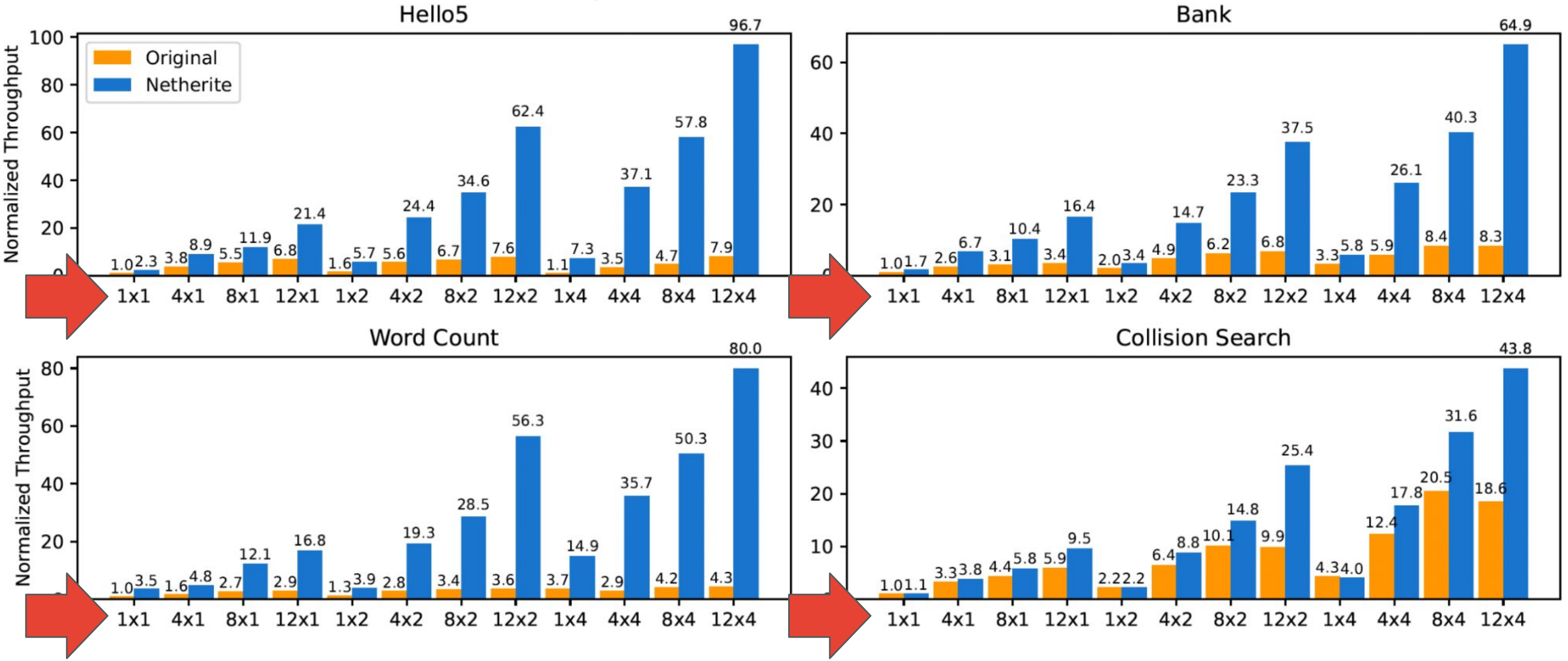
Azure Elastic Premium plans ✅

Fixed number of hosts ✅

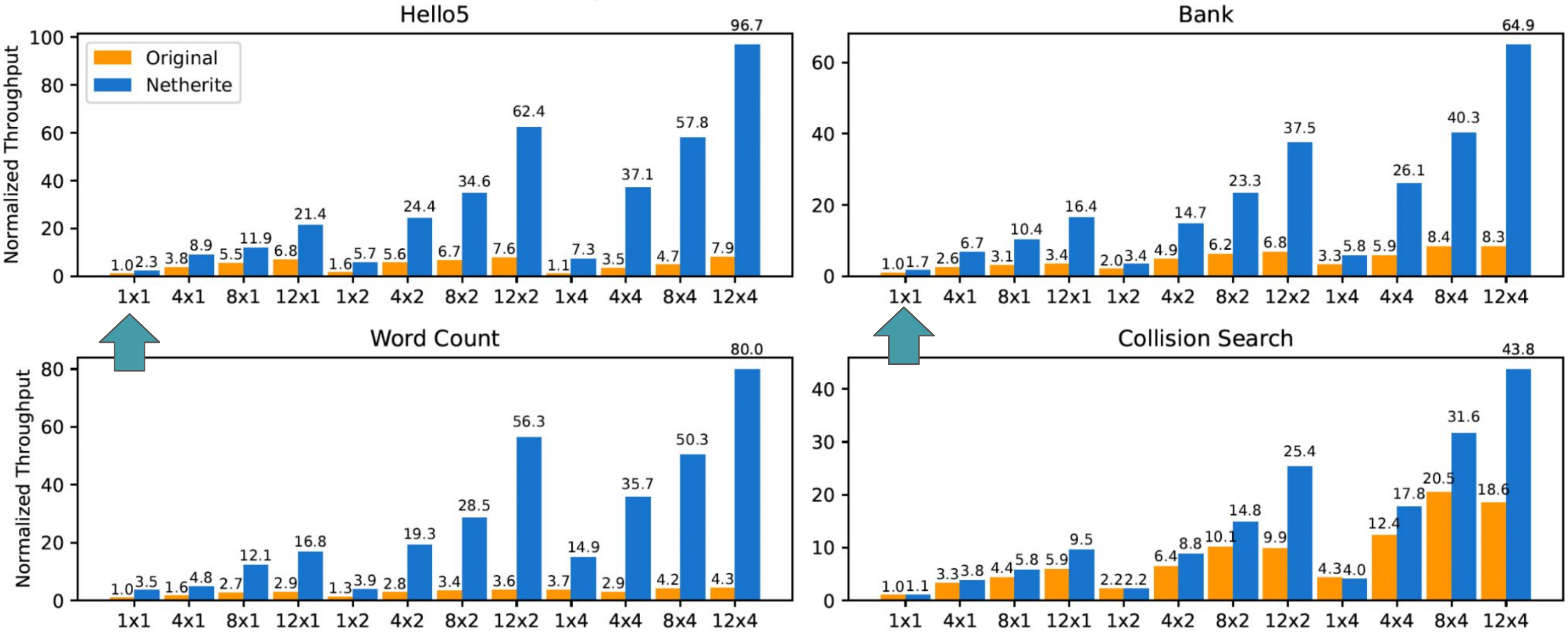
EVALUATION: Throughput Experiments

Does Netherite improve throughput compared to the original DF implementation?

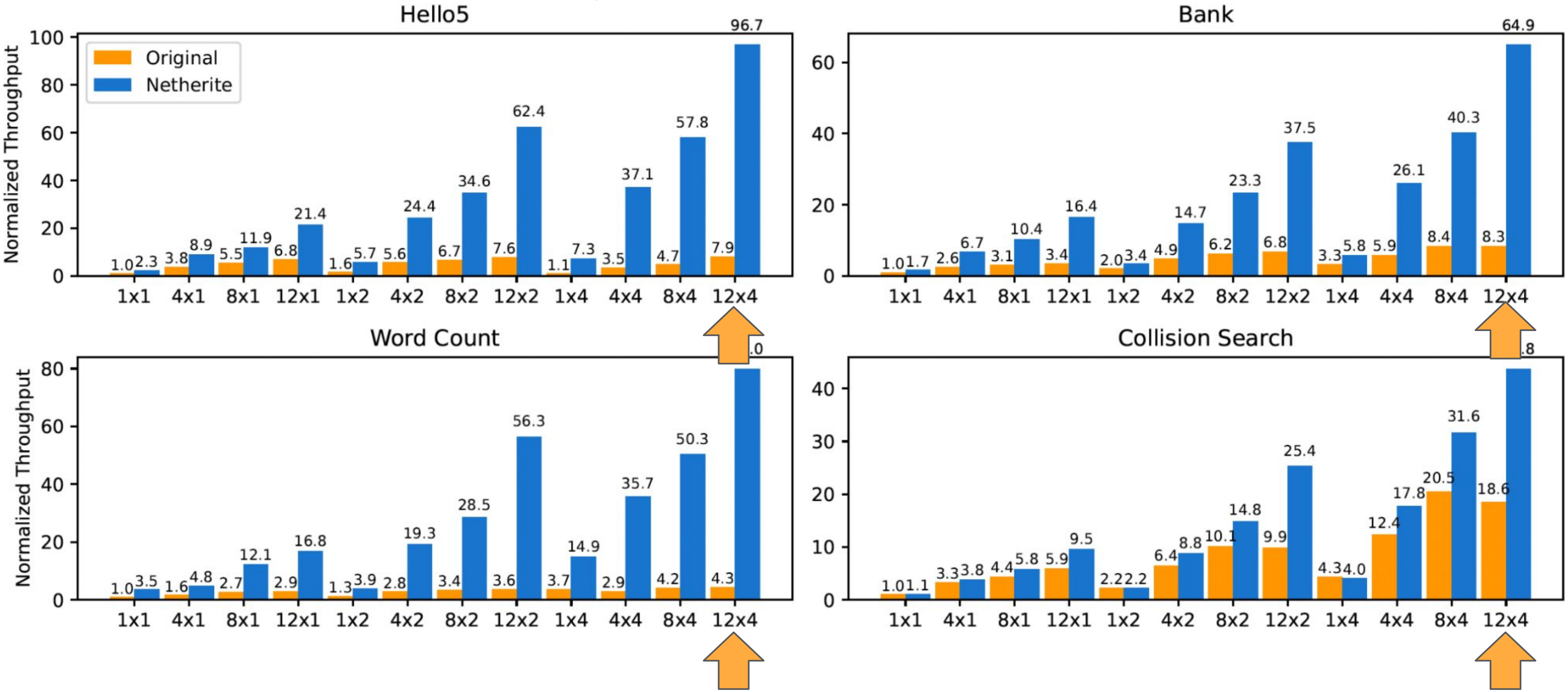
EVALUATION: Throughput Experiments



EVALUATION: Throughput Experiments



EVALUATION: Throughput Experiments



EVALUATION: Throughput Experiments

- Pipelining does not significantly affect the throughput.

EVALUATION: Throughput Experiments

- Pipelining does not significantly affect the throughput.

Pipelining is a latency optimization
so it does not reduce the amount of work

EVALUATION: Storage Traffic

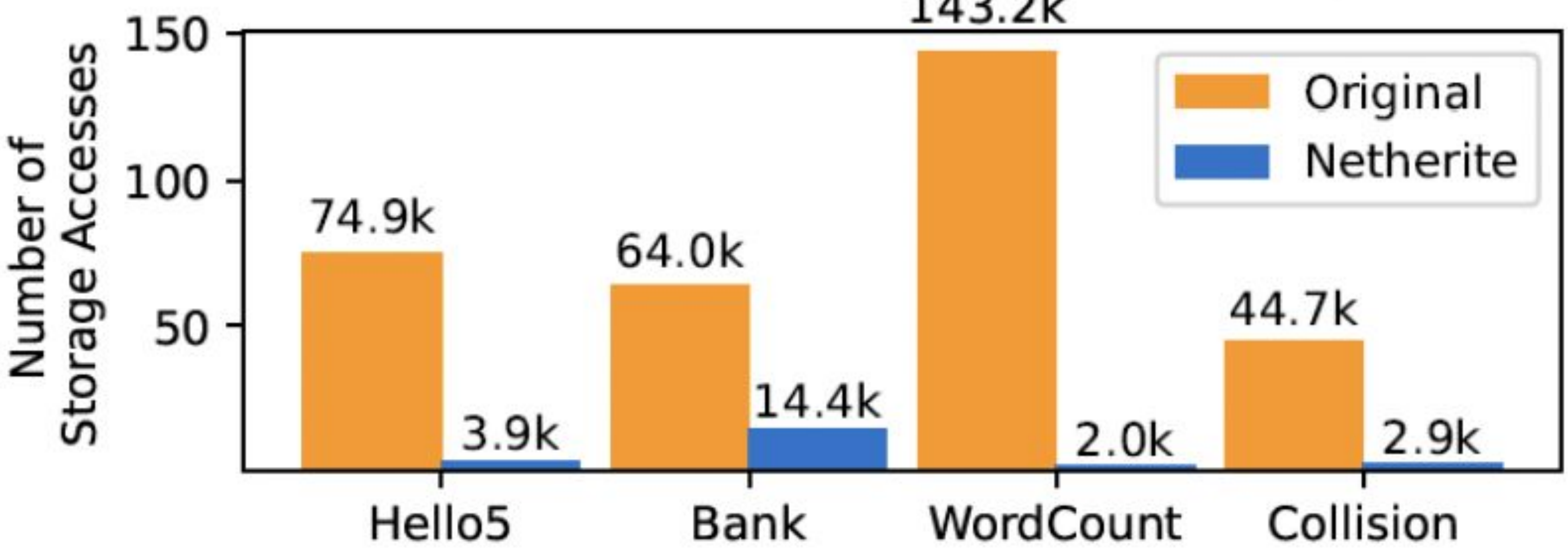
Does Netherite reduce storage traffic compared to the original DF implementation?

The Number of Storage Request

Storage Access Volume

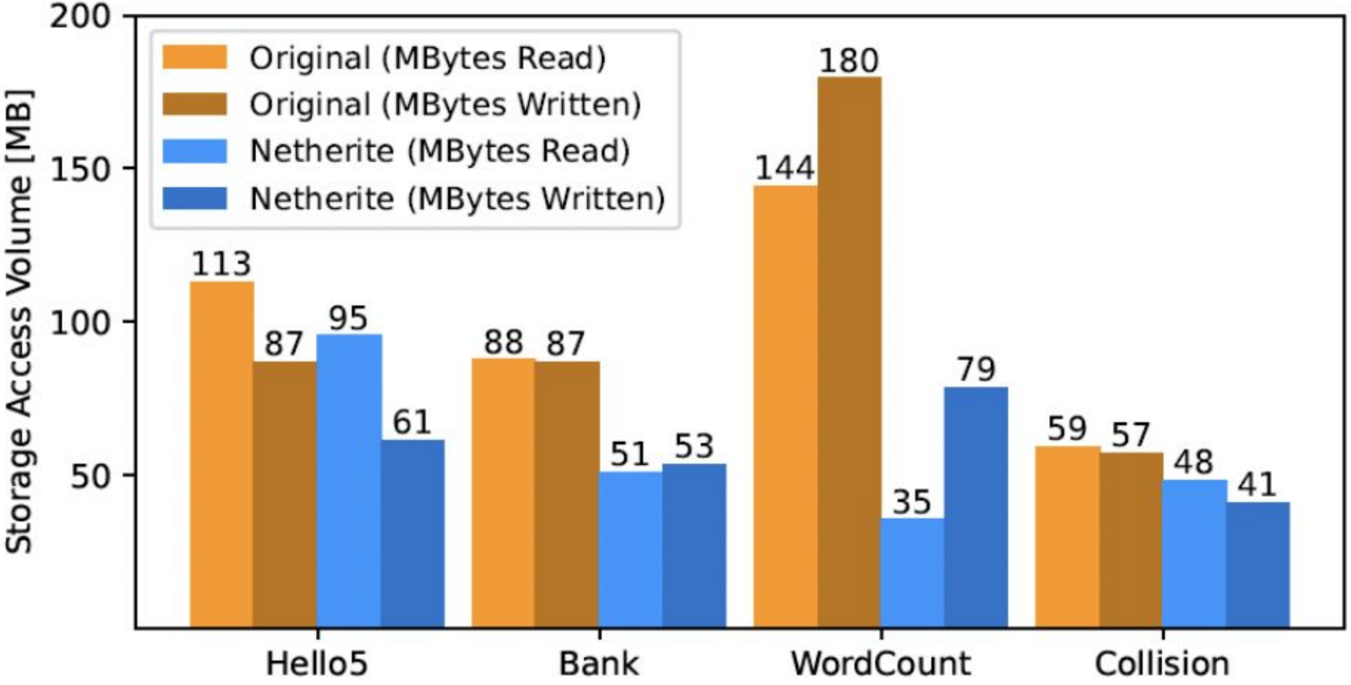
EVALUATION: Storage Traffic

Batching

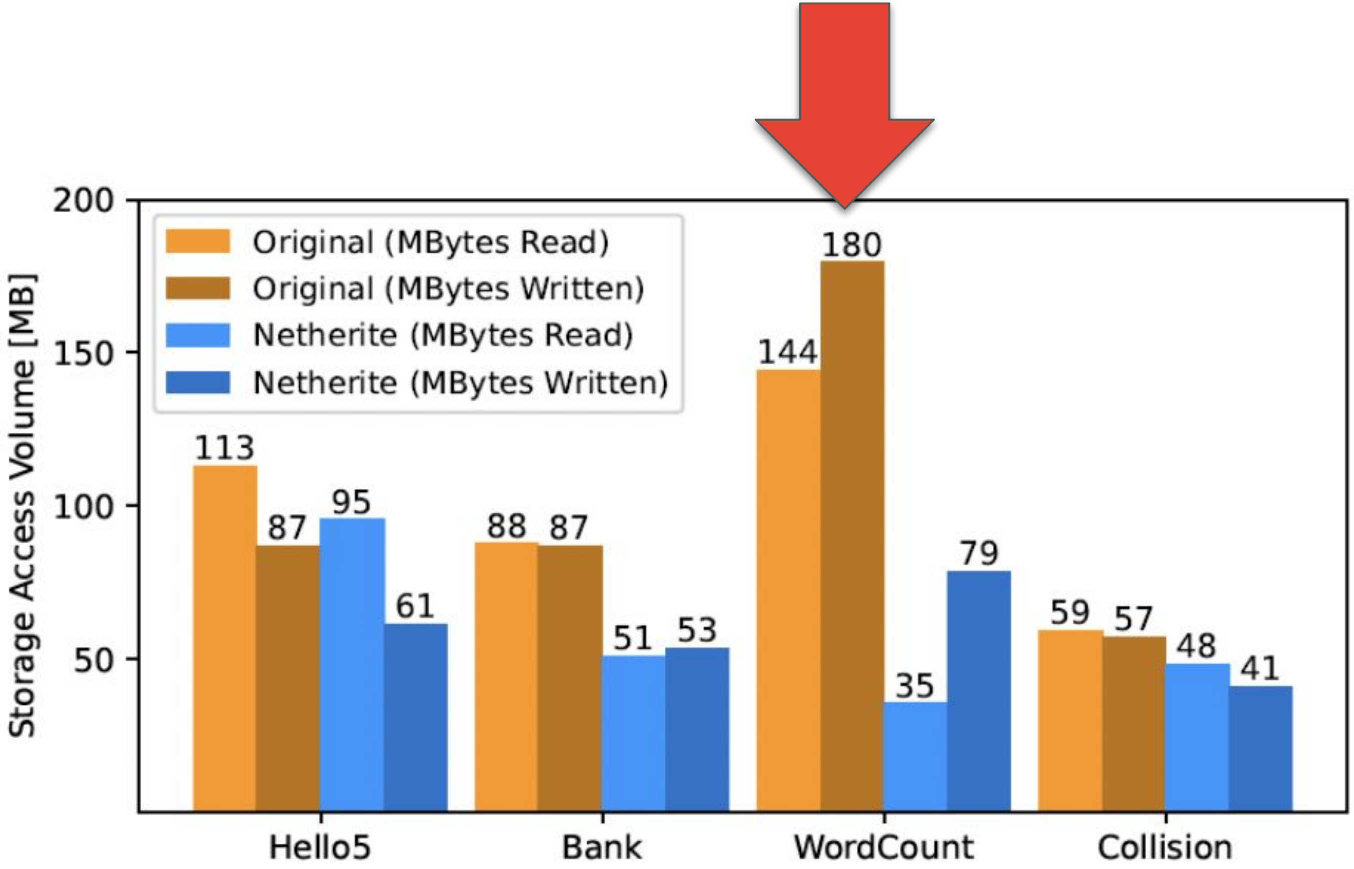


EVALUATION: Storage Traffic

Compact format



EVALUATION: Storage Traffic



EVALUATION: DF alternatives

How does Netherite compare with DF alternatives when considering the latency of a single workflow?

1. Composition
2. Step Function
3. Original Durable Function

EVALUATION: DF alternatives

How does Netherite compare with DF alternatives when considering the latency of a single workflow?

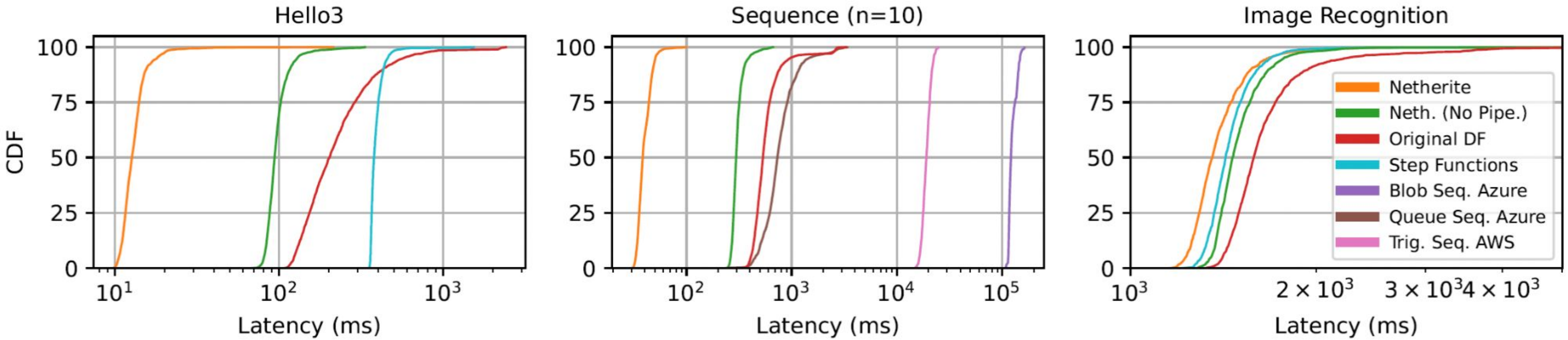
Latency Only

EVALUATION: DF alternatives

Experiment with applications:

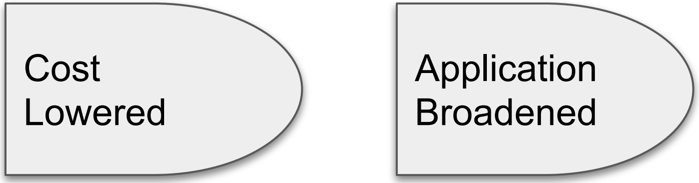
- Hello3
- Sequence
- Image Recognition

EVALUATION: Comparison to Common Alternatives



Conclusion

- Efficiency Improved …



Cost Lowered

Application Broadened